Temporal Difference Learning of N-Tuple Networks for the Game *2048*

Marcin Szubert, Wojciech Jaśkowski
Institute of Computing Science, Poznan University of Technology, Poznań, Poland
{mszubert,wjaskowski}@cs.put.poznan.pl

***Abstract*—The highly addictive stochastic puzzle game *2048* has recently invaded the Internet and mobile devices, stealing countless hours of players' lives. In this study we investigate the possibility of creating a game-playing agent capable of winning this game without incorporating human expertise or performing game tree search. For this purpose, we employ three variants of temporal difference learning to acquire i) *action value*, ii) *state value*, and iii) *afterstate value* functions for evaluating player moves at 1-ply. To represent these functions we adopt n-tuple networks, which have recently been successfully applied to Othello and Connect 4. The conducted experiments demonstrate that the learning algorithm using afterstate value functions is able to consistently produce players winning over 97% of games. These results show that n-tuple networks combined with an appropriate learning algorithm have large potential, which could be exploited in other board games.**

Keywords: temporal difference learning, n-tuple networks, game *2048*, position evaluation function, reinforcement learning

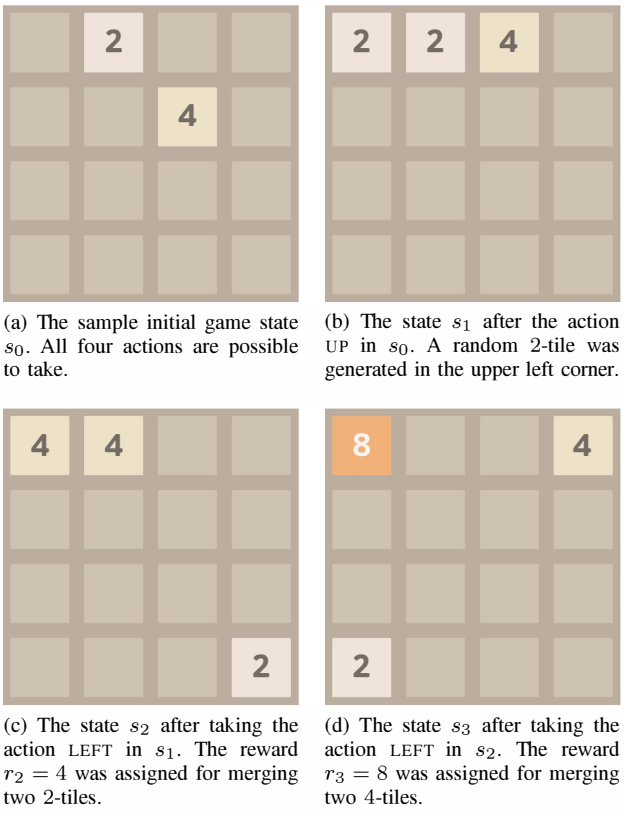
I. Introduction

The puzzle game *2048* is an addictive single-player, nondeterministic video game, which has taken the Internet by storm. According to its author, Gabriele Cirulli, during the first three weeks after the release, people spent a total time of over 3000 years on playing the game. Besides the official online version[1] and its numerous browser-based variants, clones of *2048* have reached the top of the most downloaded mobile applications rankings[2]. One of the reasons of the game's massive popularity is that it is very easy to learn but hard to master. Indeed, the game's author admitted that out of hundreds million of games ever played only about 1% have been won. The difficulty of the game together with the simplicity of its rules makes it an interesting testbed for artificial intelligence methods.

To the best of our knowledge, the only artificial intelligence techniques applied so far to the game *2048* involve game tree search with manually designed heuristic evaluation functions, which are based on human analysis of the game properties [1]. Here we ask the question whether a position evaluation function can be developed *without the use of expert knowledge*. Moreover, we are interested in such a function that is sufficiently accurate to be successfully used at 1-ply, i.e., without performing game tree search.

To investigate this issue, we compare three temporal difference learning methods applied to develop a position evaluation function represented by systematic n-tuple networks [2]. These methods provide a knowledge-free way of elaborating game-playing strategies [3]. Experimental results show that the strategies obtained in this way are not only extremely fast at making moves, but also able to win more than 97% of games. Such performance is higher than that achieved by a minimax algorithm with alpha-beta pruning [1].

[1] http://gabrielecirulli.github.io/2048/
[2] *2048* itself is a derivative of the games *1024* and *Threes*.



(a) The sample initial game state s_0. All four actions are possible to take.

(b) The state s_1 after the action UP in s_0. A random 2-tile was generated in the upper left corner.

(c) The state s_2 after taking the action LEFT in s_1. The reward $r_2 = 4$ was assigned for merging two 2-tiles.

(d) The state s_3 after taking the action LEFT in s_2. The reward $r_3 = 8$ was assigned for merging two 4-tiles.

Figure 1: A sample sequence of initial states and actions.

II. Game *2048*

2048 is a single-player, nondeterministic, perfect information video game played on a 4×4 board. Each square of the board can either be empty or contain a single v-tile, where v is a positive power of two and denotes the value of a tile. The game starts with two randomly generated tiles. Each time a random tile is to be generated, a 2-tile (with probability $p_2 = 0.9$) or a 4-tile ($p_4 = 0.1$) is placed on an empty square of the board. A sample initial state is shown in Fig. 1a.

978-1-4799-3547-5/14/$31.00 ©2014 IEEE

The objective of the game is to slide the tiles and merge the adjacent ones in order to ultimately create a tile with the value of 2048. At each turn the player makes a move consisting in sliding all the tiles in one of the four directions: UP, RIGHT, DOWN or LEFT. A move is legal if at least one tile is slid. After each move, a new 2-tile or 4-tile is randomly generated according to the aforementioned probabilities. For instance, Fig. 1b illustrates that after sliding UP both initial tiles, a random 2-tile is placed in the upper left corner.

A key operation that allows to obtain tiles with increasingly larger values consists in merging adjacent tiles. When making a move, each pair of adjacent tiles of the same value is combined into a single tile along the move direction. The new tile is assigned the total value of the two joined tiles. Additionally, for each such merge, the player gains a reward equal to the value of the new tile. Figure 1c shows the board resulting from combining two 2-tiles in the upper row when sliding the tiles in the LEFT direction. Another LEFT move leads to creating an 8-tile (see Fig. 1d). The moves generate rewards of 4 and 8, respectively.

The game is considered won when a 2048-tile appears on the board. However, players can continue the game even after reaching this tile. The game terminates when there are no legal moves, i.e., all squares are occupied and there are no two adjacent tiles sharing the same value. The game score is the sum of rewards obtained throughout the game.

III. METHODS

In this section we discuss how to create a game-playing agent capable of winning the game *2048* without incorporating human expertise. Our approach is based on temporal difference learning (TDL, [4]), a widely applied class of reinforcement learning [5] methods. This approach enables an agent to autonomously learn sequential decision making just by trial-and-error runs in the environment framed as a Markov decision process [6].

A. Markov Decision Processes

The framework of Markov decision processes (MDPs) allows to model problems in which an agent must perform a sequence of actions in the given environment. Actions change the state of the environment and typically result in both immediate and delayed consequences that can be quantified as rewards for the agent.

Formally, an MDP is a discrete time stochastic control process, and can be defined as $\langle S, A, R, P \rangle$, where:

- S is a set of possible states of the environment,
- A is a set of actions and $A(s) \subseteq A$ denotes a set of actions available in state $s \in S$,
- $R : S \times A \to \mathbb{R}$ is a reward function, where $R(s, a)$ provides the reward for making action a in state s,
- $P : S \times A \times S \to [0, 1]$ is a stochastic transition function, where $P(s, a, s'')$ denotes the probability of transition to state s'' in result of taking action a in state s.

The objective of an agent placed in an environment defined as an MDP is to learn such a decision making policy $\pi : S \to A$ that maximizes the expected cumulative reward.

The game *2048* can be naturally formulated as an MDP, in which states are board positions and actions are legal moves. Both the reward function and the transition function are clearly specified by the game rules. Importantly, the transition function is nondeterministic due to the random tile generation procedure, which determines the probabilities $P(s, a, s'')$.

B. Temporal Difference Learning

Since the influential work of Tesauro [7] and the success of his TD-Gammon player learned through self-play, TDL has become a well-known approach for elaborating game-playing agents with little or no help from human designer or expert strategies given *a priori*. Although TDL was introduced by Sutton [4], its origins reach back to the famous checkers playing program developed by Samuel [8]. Nevertheless, it was TD-Gammon that has triggered off extensive research on using TDL to learn policies for such games as Go [9], [10], Othello [11], [12], and Chess [13].

As an intermediate step towards learning decision making policies, TDL methods compute value functions. The state value function $V^\pi : S \to \mathbb{R}$ estimates the expected cumulative reward that will be received by the agent if it uses policy π to make actions starting from a given state $s \in S$. In the context of playing games, the state value function predicts how much points the agent will get from the given state till the end of the game.

To learn the state value function, the agent uses experience from interactions with the environment. Whenever the agent takes an action a, observes a state transition $s \to s''$ and receives a reward r, it updates the current estimate of $V(s)$. In particular, the simplest TDL algorithm [4], known as TD(0), employs the following update rule:

$$V(s) \leftarrow V(s) + \alpha(r + V(s'') - V(s)). \tag{1}$$

This rule attempts to minimize the difference between the current prediction of cumulative future reward $V(s)$ and the one-step-ahead prediction, which involves the actual (received) reward r and is equal to $r + V(s'')$. Consequently, the error between the successive predictions $\delta = r + V(s'') - V(s)$ is used to adjust the value of state s. The size of correction is determined by the learning rate $\alpha \in [0, 1]$.

For some problems, it might be more effective to learn the action value function $Q^\pi : S \times A \to \mathbb{R}$, which estimates the utility of a given state-action pair, i.e., the expected sum of future rewards after making action $a \in A(s)$ in state $s \in S$. One of the most recognized algorithms for learning action value functions is Q-LEARNING [14], which operates in a similar way to TD(0). Upon observing a transition (s, a, r, s''), it updates the action value estimates as follows:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \max_{a' \in A(s'')} Q(s'', a') - Q(s, a)). \tag{2}$$

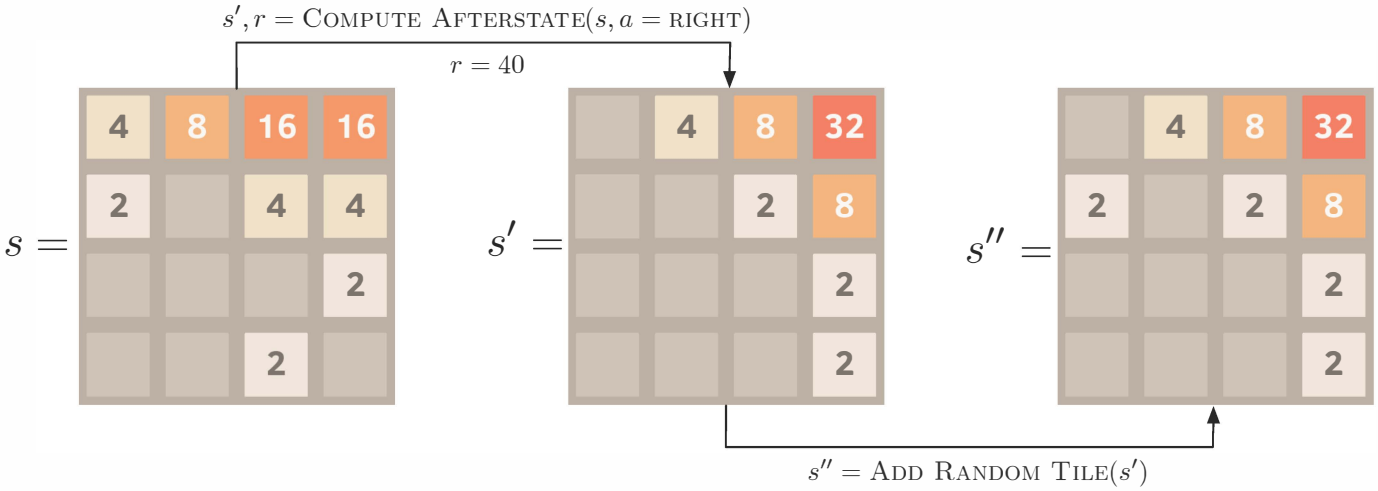


Figure 2: A two-step state transition occurring after taking the action $a =$ RIGHT in the state s.

C. Learning Game-Playing Policies

In this section we describe how both TDL algorithms, namely TD(0) and Q-LEARNING, can be employed to develop a game-playing policy for the game *2048*. Figure 3 presents the pseudocode of the game engine used throughout this study to play and learn the game. The crucial part of the game mechanics is implemented by the function MAKE MOVE, which for a given state $s \in S$ and action $a \in A(s)$ returns a received reward and an observed state transition.

Importantly, in the case of *2048* the state transition can be regarded as a two-step process, which is illustrated in Fig. 2. Firstly, the deterministic *afterstate* s' and the reward r are computed by sliding and merging the tiles according to the selected action a. Secondly, a random tile is added to the afterstate s' to create the next state s'', in which the agent will make its next move. We will exploit the notion of afterstates in one of the employed learning algorithms.

The actual game-playing policy is determined by the function EVALUATE, which attempts to measure the utility of taking each possible action $a \in A(s)$ in the current state s. Moves are selected to maximize the value returned by this function. Consequently, learning a policy consists in adjusting this function in order to make more accurate moves evaluation.

In the following subsections we consider three different ways of implementing the EVALUATE function. Each of them requires a dedicated LEARN EVALUATION algorithm, which adjusts the evaluation function on the basis of the observed experience represented by a tuple (s, a, r, s', s'').

1) Evaluating actions: A straightforward approach to implement the EVALUATE(s, a) function is to employ directly the action value function $Q(s, a)$. In such case the agent takes actions specified by the following policy:

$$\pi(s) = \arg\max_{a \in A(s)} Q(s, a).$$

Since the game *2048* involves only four actions, instead of a single Q-function we can maintain four state value functions — a separate function V_a for each action $a \in A$. This approach

```
1:  function PLAY GAME
2:      score ← 0
3:      s ← INITIALIZE GAME STATE
4:      while ¬IS TERMINAL STATE(s) do
5:          a ← arg max_{a'∈A(s)} EVALUATE(s, a')
6:          r, s', s'' ← MAKE MOVE(s, a)
7:          if LEARNING ENABLED then
8:              LEARN EVALUATION(s, a, r, s', s'')
9:          score ← score + r
10:         s ← s''
11:     return score
12:
13: function MAKE MOVE(s, a)
14:     s', r ← COMPUTE AFTERSTATE(s, a)
15:     s'' ← ADD RANDOM TILE(s')
16:     return (r, s', s'')
```

Figure 3: A pseudocode of a game engine with moves selected according to the evaluation function. If learning is enabled, the evaluation function is adjusted after each move.

```
1: function EVALUATE(s, a)
2:     return V_a(s)
3:
4: function LEARN EVALUATION(s, a, r, s', s'')
5:     v_next ← max_{a'∈A(s'')} V_{a'}(s'')
6:     V_a(s) ← V_a(s) + α(r + v_next − V_a(s))
```

Figure 4: The *action evaluation function* and Q-LEARNING.

is illustrated in Fig. 1. The functions are learned according to the Q-LEARNING update rule (cf. Equation 2).

2) Evaluating states: An alternative approach to evaluating moves is to assess the states in which they result with the state value function $V(s)$. In contrast to evaluating actions (cf. Fig. 1), this approach requires from the agent to know the environment model, i.e., the reward function R and the

```
function EVALUATE(s, a)
    s', r ← COMPUTE AFTERSTATE(s, a)
    S'' ← ALL POSSIBLE NEXT STATES(s')
    return r + Σ_{s''∈S''} P(s, a, s'')V(s'')

function LEARN EVALUATION(s, a, r, s', s'')
    V(s) ← V(s) + α(r + V(s'') − V(s))
```

Figure 5: The *state evaluation function* and TD(0).

```
function EVALUATE(s, a)
    s', r ← COMPUTE AFTERSTATE(s, a)
    return r + V(s')

function LEARN EVALUATION(s, a, r, s', s'')
    a_next ← arg max_{a'∈A(s'')} EVALUATE(s'', a')
    s'_next, r_next ← COMPUTE AFTERSTATE(s'', a_next)
    V(s') ← V(s') + α(r_next + V(s'_next) − V(s'))
```

Figure 6: The *afterstate evaluation function* and a dedicated variant of the TD(0) algorithm.

transition function P. If this requirement is satisfied, the agent can use the following policy to select actions:

$$\pi(s) = \arg\max_{a \in A(s)} \left[R(s,a) + \sum_{s'' \in S} P(s,a,s'')V(s'') \right].$$

The corresponding EVALUATE function is shown in Fig. 5. Noteworthy, evaluating a move requires computing all possible states resulting from this move and applying the state value function to each of them. In the case of the game *2048* there may be as many as 30 different transitions that may result from making a single move (there are up to 15 empty squares in the afterstate, and two different types of tiles may be placed on each square). For this reason, when compared with the action evaluation scheme, which requires only a single calculation of the V-function, this approach is expected to be significantly slower. On the other hand, the learning procedure is simple and consists in adjusting the state value function with the TD(0) update rule (cf. Equation 1).

3) Evaluating afterstates: The last considered approach to evaluate moves can be regarded as a combination of the action evaluation and the state evaluation. This approach uses a single value function V, but instead of applying it to states, as in the case of the state evaluation, it assesses the values of afterstates. Since for each move there is only a single deterministic afterstate, the evaluation is almost as fast as in the case of the action evaluation. In this approach, the agent takes actions according to the following policy:

$$\pi(s) = \arg\max_{a \in A(s)} \left[R(s,a) + V\left(T(s,a)\right) \right],$$

where $T(s,a)$ denotes the mapping from state s and action a to the resulting afterstate s'. Figure 6 presents the afterstate evaluation function. The learning procedure involves computing the error between two subsequent afterstates. Therefore, it starts with determining the next action that would be taken by the agent in the new state $a_{next} = \pi(s'')$ and uses it to compute the next reward r_{next} and the new afterstate $s'_{next} = T(s'', a_{next})$. Knowing s', s'_{next} and r_{next} it updates the value of the recently observed afterstate s'.

D. N-tuple Network Evaluation Function

The major design choice concerns the representation of the value functions $V : S \to \mathbb{R}$, which are used by each of the considered algorithms to evaluate moves and, in effect, determine the game-playing policy. In problems with small state spaces, the value function V can be represented directly, as a look-up table, with each value stored individually. The game *2048* has ca. $(4 \times 4)^{18} \approx 4.7 \times 10^{21}$ states because the maximum value of a single tile[3] is $2^{17} = 131072$. Therefore, using an explicit value table is computationally infeasible. Instead, we have to use a function approximator that adopts a class of parameterized functions to replace the value table. One particular type of such approximators are *n-tuple networks* [15], which have been recently successfully applied to Othello [16], [17], [18], [12] and Connect 4 [19].

An n-tuple network consists of m n_i-tuples, where n_i is tuple's size. For a given board state s, it calculates the sum of values returned by the individual n-tuples. The ith n_i-tuple, for $i = 1 \dots m$, consists of a predetermined sequence of board locations $(loc_{ij})_{j=1 \dots n_i}$, and a look-up table LUT_i. The latter contains weights for each board pattern that can be observed on the sequence of board locations. Thus, an n-tuple network implements a function f:

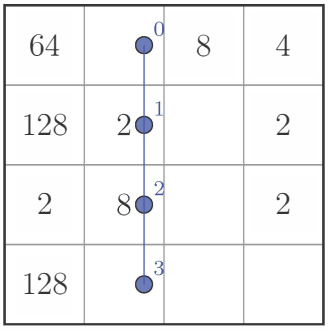
$$\begin{aligned} f(s) = \sum_{i=1}^{m} f_i(s) &= \sum_{i=1}^{m} \text{LUT}_i \left[\text{index}\left(s_{loc_{i1}}, \dots, s_{loc_{in_i}}\right) \right], \\ \text{index}\left(\mathbf{v}\right) &= \sum_{j=1}^{|\mathbf{v}|} v_j c^{j-1}, \end{aligned}$$

where $s_{loc_{ij}}$ is a board value at location loc_{ij}, $\mathbf{v}$ is a sequence of board values (the observed pattern) such that $0 \le v_j < c$, for $j = 1 \dots |\mathbf{v}|$, and c is a constant denoting the number of possible board values. As we have already pointed out, theoretically c is equal to 18 in the game *2048*, but in order to limit the number of weights in the network, in the experiments we assume $c = 15$. An empty square is encoded as 0, while a square containing a value v as $\log_2 v$, e.g., 128 is encoded as 7. See Fig. 7 for an illustration.

IV. COMPARISON OF THE LEARNING METHODS

In the first experiment, we compared the performance of the three learning methods described in Section III-C. To provide fair comparison, each method used the same number of training games equal to 500 000. For statistical significance, each experimental run was repeated 30 times.

[3]Practically, we are not aware of any credible reports of obtaining a tile with a value higher than $2^{14} = 16384$.



0123	weight
0000	3.04
0001	−3.90
0002	−2.14
⋮	⋮
0010	5.89
⋮	⋮
0130	**-2.01**
⋮	⋮

Figure 7: A straight 4-tuple on a *2048* board. According to the values in its lookup table, for the given board state it returns -2.01, since the empty square is encoded as 0, the square containing 2 as 1, and the square containing 8 as 3.

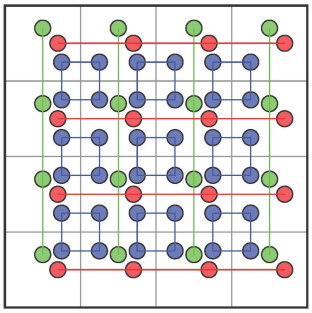


Figure 8: The n-tuple network consisting of all possible horizontal and vertical straight 4-tuples (red and green, respectively), and all possible 2×2 square tuples (blue).

A. Experimental Setup

1) N-tuple network architecture: An important design issue of an n-tuple network architecture is the location of individual n-tuples on the board [12]. Although this is not a formal requirement, due to the spatial nature of game boards, n-tuples are usually consecutive sequences of locations [15][16].

For the game *2048*, a single n-tuple involves 15^n weights. Thus, in order to obtain networks with a manageable number of weights, we limited the length of tuples to 4. We used a systematic n-tuple network [2] consisting of 4 horizontal and 4 vertical 4-tuples, and 9 'square' 4-tuples (see Fig. 8). In total, a network contained 17 4-tuples and thus involved $17 \times 15^4 = 860\,625$ weights.

2) Learning algorithms: All the learning methods employed a single learning agent with the evaluation function represented by n-tuple networks. As presented in Fig. 3, the evaluation function was used to assess the utility of possible moves and thus determined the game-playing policy. However, each of the following setups used a different way of evaluating the move and learning the evaluation function values.

1) Q-LEARNING. The evaluation function consists of four n-tuple networks that provide value functions for each of the possible game moves: V_{UP}, V_{RIGHT}, V_{DOWN}, V_{LEFT}. The move evaluation involves evaluating each action independently, while learning was realized by the Q-LEARNING algorithm (cf. Fig. 1).
2) TD-STATE. A single n-tuple network acts as the state value function $V(s)$. It is used to evaluate all possible states resulting from the given move. The move evaluation requires knowledge of the transition probabilities and the reward function. The learning algorithm is straightforward TD(0) as shown in Fig. 5.
3) TD-AFTERSTATE. A single n-tuple network is employed as the afterstate value function $V(s')$ to assess the deterministic results of making the given action. The learning step is performed according to the TD(0) algorithm adopted to operate on afterstates (cf. Fig. 6). Both the move evaluation and value function learning require the game model to compute afterstates. On the other hand, when compared with TD-STATE, the transition probabilities are not needed.

In each of the considered methods, learning is performed after each move and consists in adjusting the values of the maintained V-functions for particular states or afterstates. Since each such function is implemented by an n-tuple network, the adjustment of the form $V(s) \leftarrow V(s) + \alpha\,(V(s') - V(s))$ implies the following change of the LUT weights for each tuple $i = 1 \ldots m$:

$$\Delta\text{LUT}_i\left[\text{index}\left(s_{loc_{i1}}, \ldots, s_{loc_{in_i}}\right)\right] = \alpha\,(f(s') - f(s))\,.$$

In all the experiments, the weights were initially set to 0. The only parameter of the learning methods is the learning rate α — we consider five different values of this parameter, $\alpha \in \{0.001, 0.0025, 0.005, 0.0075, 0.01\}$.

3) Performance measures: To evaluate the learning agents we used the following performance measures:

1) *Winning rate* $\in [0, 1]$ — the fraction of games the agent won (i.e., reached a 2048-tile) over 1000 games played,
2) *Total score*, — the total number of points obtained during the game averaged over 1000 games,
3) *Learning time* — the CPU time spent on learning the agent.

Algorithm	Best winning rate	Best total score	CPU time [s]
Q-LEARNING	0.4980 ± 0.0078	20504.6 ± 163.5	3136.8 ± 61.7
TD-STATE	0.8672 ± 0.0122	48929.6 ± 702.5	24334.7 ± 405.7
TD-AFTERSTATE	0.9062 ± 0.0051	51320.9 ± 358.4	7967.5 ± 165.3

Table II: Summary of the results obtained after 500 000 training games by the three temporal difference learning methods.

B. Results

The results achieved by the the three compared learning methods with different learning rate settings are shown in Table I. The first observation is that all the methods are capable of producing players winning the game *2048* at least from time to time. However, the differences in performance achieved by particular methods are significant. Their winning rate ranges from 0.16 to 0.90, while the average total score varies from 15 322 to 51 320.

Learning rate	Winning rate			Total score		
	Q-LEARNING	TD-STATE	TD-AFTERSTATE	Q-LEARNING	TD-STATE	TD-AFTERSTATE
0.0010	0.1672 ± 0.0262	0.8622 ± 0.0059	0.8821 ± 0.0068	15322.34 ± 322.84	44607.34 ± 972.77	49246.74 ± 563.83
0.0025	0.4796 ± 0.0058	$\mathbf{0.8672 \pm 0.0122}$	$\mathbf{0.9062 \pm 0.0051}$	20453.40 ± 122.38	$\mathbf{48929.66 \pm 702.47}$	$\mathbf{51320.93 \pm 358.42}$
0.0050	$\mathbf{0.4980 \pm 0.0078}$	0.8660 ± 0.0120	0.8952 ± 0.0089	$\mathbf{20504.66 \pm 163.48}$	46838.72 ± 578.18	48939.73 ± 722.61
0.0075	0.4658 ± 0.0090	0.8253 ± 0.0131	0.8867 ± 0.0077	19804.91 ± 226.34	43384.73 ± 696.03	46423.26 ± 619.72
0.0100	0.4438 ± 0.0103	0.8083 ± 0.0170	0.8601 ± 0.0090	19314.17 ± 247.56	41114.58 ± 523.70	43387.24 ± 541.70

Table I: Average performances of learning agents after 500 000 training games. '$\pm$' precedes half of the width of the 95% confidence interval.

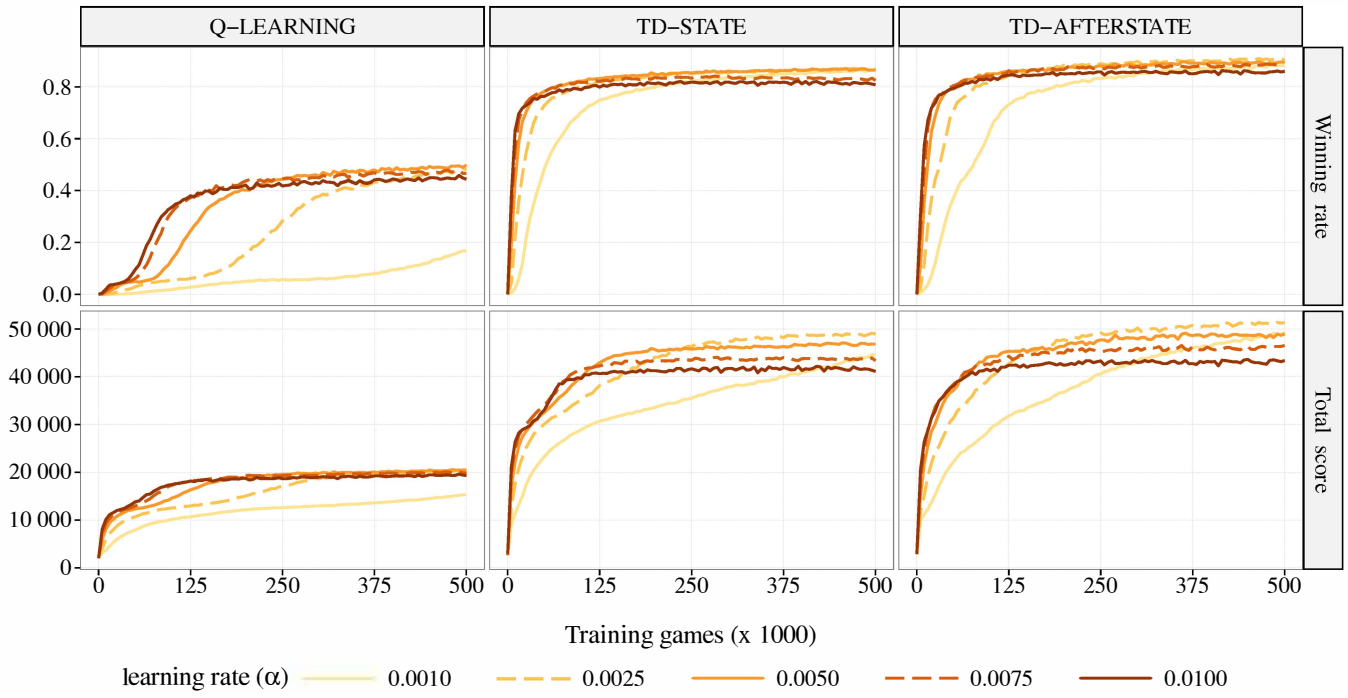


Figure 9: Performance of the three learning methods with different learning rates as a function of the number of training games.

The best overall performance is obtained by TD-AFTERSTATE, which is slightly, but statistically significantly (t-test with confidence level equal to 0.01), better than TD-STATE. Although the gap between these two methods is minor, TD-AFTERSTATE works substantially faster (see Table II). Let us remind from Section III-C that it is because TD-STATE evaluates all possible transitions that may result from making a single move, whereas TD-AFTERSTATE considers only a single deterministic afterstate. Moreover, Table I shows also that Q-LEARNING is significantly worse than both TD-STATE and TD-AFTERSTATE regardless of the learning rate and the considered performance measure.

To gain more insight into the learning process, we plotted the performance of learning agents as a function of the number of training games (see Fig. VI-A). We can see that the curves of Q-LEARNING have not yet converged after 500 000 training games. Although we do not know whether given enough time Q-LEARNING would finally catch up with the other two methods, it is clear that even for the best found learning rate ($\alpha = 0.005$), it learns slower than its competitors.

Figure VI-A reveals also how the learning rate parameter influences the learning progress. Although higher learning rates make learning quicker in the beginning, lower values are better in the long run. For TD-STATE and TD-AFTERSTATE $\alpha = 0.0025$ provided the best results, but, according to the figure, it is possible that $\alpha = 0.001$ would surpass it during successive few hundred thousands training games.

V. Improving the Winning Rate

In order to improve the winning rate of produced game-playing policies, we took the lessons learned in the previous section and conducted another experiment using more computational resources.

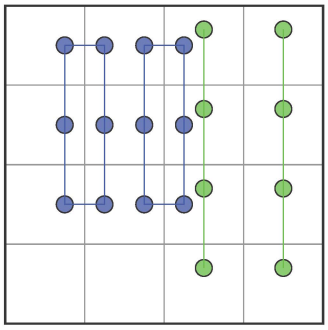


Figure 10: A network consisting of all possible 2×3 rectangle tuples (blue) and all possible straight 4-tuples (green). This network makes use of board symmetry (symmetric sampling), thus only two n-tuples of each kind are used.

A. Experimental Setup

Since the previous results indicate that TD-AFTERSTATE with $\alpha = 0.0025$ is the most effective learning setup, in this experiment we used solely this method, letting it to train the agent for 1 million games.

Additionally, to increase the capabilities of the evaluation function, we improved the architecture of the n-tuple network. The improvement was twofold. First, we used larger tuples — apart from straight 4-tuples we added rectangular-shaped 6-tuples (see Fig. 10). Second, we reduced the number of weights and, at the same time, improved generalization of the n-tuple network by using *symmetric sampling*, which exploits the symmetries of the *2048* board. In symmetric sampling, a single n-tuple is employed eight times, returning one value for each possible board rotation and reflection. In this way, we needed only 2 straight 4-tuples and 2 rectangular-shaped 6-tuples as depicted in Fig 10. This network contained $2 \times 15^4 + 2 \times 15^6 = 22\,882\,500$ weights — two orders of magnitude more than the number of weights of the network used in Section IV.

B. Results

Figure 11 shows the performance of the learning agent for two types of networks: i) the *small,* standard one, introduced in Section IV, and ii) the *large*, symmetric one, shown in Fig. 10. Clearly, using the large network results in a higher performance. After one million training games, the agents equipped with the large networks win 97.0% of games on average, compared with 91.6% of games won by the players using the small one. The performance gap between the networks is even more visible when we compare the average total scores. The large network allowed to obtain the score of $99\,916 \pm 1290$ on average, which is nearly two times more than that obtained with the small network ($52\,172 \pm 369$).

Let us also analyze how learning of the large network progresses over time in terms of the average total score. In the bottom plot of Fig. 11 we can see that after around 100 000 training games, the speed of learning gradually decreases to start increasing again 200 000 training games later. We

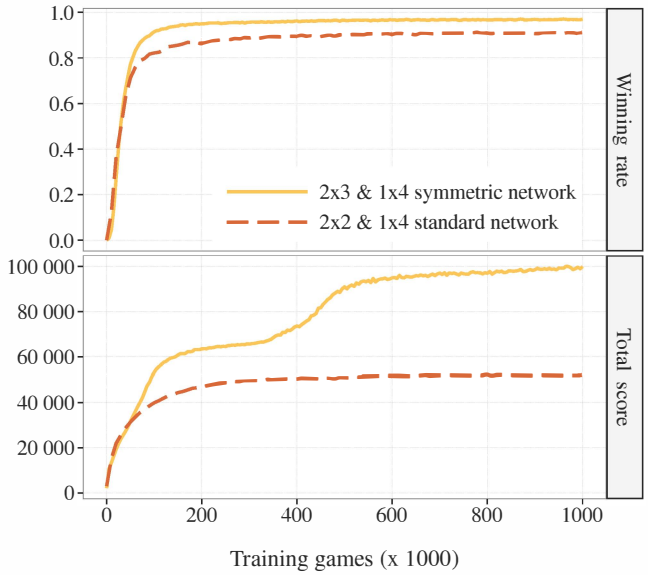


Figure 11: TD-AFTERSTATE learning applied to two types of n-tuple network architectures.

performed a brief analysis to explain these 'inflection points' on the learning curve and we found out that after around 350 000 training games, the agents learned how to build 16 384-tiles. This resulted in substantially higher rewards, and caused the observed increase in learning speed.

Finally, in order to estimate the quality of the single best agent (in terms of the winning rate) found in this experiment[4], we measured its performance in 1 000 000 games. Its winning rate was 0.9781, and it scored 100 178 points on average. The highest score obtained in a single game was equal to 261 526.

VI. Discussion

A. Winning Rate vs. Total Score

The best winning rate we obtained is nearly 0.98. We suspect, however, that, even with the same evaluation function, a higher winning rate is possible. Since the goal of the agent is to maximize the sum of future rewards, it focuses on the total score performance measure. The agent has not been given any direct incentive to learn to win the game more often. Naturally, the winning rate is largely correlated with the total score, what we could see in Fig. VI-A and Fig. 11. We can observe, however, that at some point of learning, while the total score is still increasing, the winning rate stays at the same level — see the learning curves in Fig. 11, for example. This is because a rational agent prefers to get *sometimes* the reward of 16 384 rather than to get *always* the reward of 2048, which would mean winning the game. A reward scheme providing agents direct incentives for obtaining a 2048-tile would, presumably, lead to even higher winning rates.

[4]The accompanying material for this study (including the best players found in the experiments) is available online at http://www.cs.put.poznan.pl/mszubert/projects/2048.html.

B. Comparison with Search-Based Agents

Although comparing reinforcement learning with other AI techniques was not the goal of this study, it is interesting to take a rough look on the existing approaches to the game *2048*. As no published research exists about this game, we fell back to the reports available on the Internet to find two anonymously published search-based algorithms developed for *2048* [1].

The first approach relies on minimax game tree search. It uses alpha-beta pruning, a human-designed static evaluation function and several hand-tuned parameters. Its winning rate depends on the time the minimax is allowed to search for the best move. We checked that, given 100 ms for making a move, it wins 89% of games, which is a significantly worse result than that achieved by the best of our agents.

The second method employs expectimax search with the depth limit of 8 moves, transposition tables, and a heuristic evaluation function. Reportedly, it is capable of winning the game in 100% of cases, but this number is based on only 100 games, and thus it may be overestimated. Interestingly, however, despite an efficient C implementation with some low-level optimizations, a single game takes as long as 37 minutes, that is, 6.6 moves per second, on average. Although this may seem quick in absolute terms, it is approximately 50 000 times slower than our agent, which completes a single game in just 23 ms, being able to take 330 000 moves per second.

The above brief comparison shows an interesting, albeit frequently observed, trade-off between our learning-based agents and search-based agents. The former use little computational power but may require substantial amount of memory to store the evaluation function, while the latter may require little memory, but are much more computationally demanding.

C. Lack of Exploration

Let us also point out that we did not employ any explicit exploration mechanism during the learning. Although we did some preliminary experiments with ϵ-greedy exploration [5], it did not not improve the performance. The exploration is not needed in this game, as the environment is inherently stochastic and thus provides sufficiently diversified experience.

VII. Conclusions

The puzzle game *2048* constitutes a new interesting challenge for computational intelligence methods. In this study we have shown that for this stochastic game the TD-AFTERSTATE learning algorithm equipped with a powerful evaluation function (n-tuple network) is able to produce agents winning nearly 98% of games on average. This performance obtained at 1-ply is comparable with the performance of the best computationally-intensive search-based agents.

An important lesson learned from this work is that particular TDL algorithms, despite working in a similar manner, may achieve diametrically different performance. Here, the Q-LEARNING algorithm applied to learn the action value function was significantly outperformed by TD(0) algorithms. In particular, learning *afterstate* values is a viable alternative for learning *state* values in stochastic environments where the agent can compute the immediate effects of its moves, but it is difficult to obtain the entire state transition.

In a broader perspective, our results demonstrate that n-tuple networks combined with an appropriate learning algorithm can work successfully even with millions of weights. Clearly, this combination has large potential for learning game-playing policies also for other board games.

Acknowledgment

M. Szubert and W. Jaśkowski have been supported by the Polish National Science Centre grants no. DEC-2012/05/N/ST6/03152 and DEC-2013/09/D/ST6/03932.

References

[1] "What is the optimal algorithm for the game, 2048?" http://stackoverflow.com/questions/22342854, accessed: 2014-15-04.

[2] W. Jaśkowski, "Systematic n-tuple networks for position evaluation: Exceeding 90% in the othello league," 2014, http://arxiv.org/abs/1406.1509.

[3] J. Mańdziuk, *Knowledge-Free and Learning-Based Methods in Intelligent Game Playing*. Springer, 2010, vol. 276.

[4] R. S. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, 1988.

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

[6] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. NY, USA: John Wiley & Sons, Inc., 1994.

[7] G. Tesauro, "Temporal Difference Learning and TD-Gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[8] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development*, vol. 44, no. 1, pp. 206–227, 1959.

[9] T. P. Runarsson and S. M. Lucas, "Co-evolution versus Self-play Temporal Difference Learning for Acquiring Position Evaluation in Small-Board Go," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 628–640, 2005.

[10] N. N. Schraudolph, P. Dayan, and T. J. Sejnowski, "Learning to Evaluate Go Positions via Temporal Difference Methods," in *Computational Intelligence in Games*, ser. Studies in Fuzziness and Soft Computing, N. Baba and L. C. Jain, Eds. Springer Verlag, Berlin, 2001, vol. 62, ch. 4, pp. 77–98.

[11] S. van den Dries and M. A. Wiering, "Neural-Fitted TD-Leaf Learning for Playing Othello With Structured Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 11, pp. 1701–1713, 2012.

[12] M. G. Szubert, W. Jaśkowski, and K. Krawiec, "On Scalability, Generalization, and Hybridization of Coevolutionary Learning: A Case Study for Othello," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 5, no. 3, pp. 214–226, 2013.

[13] J. Baxter, A. Tridgell, and L. Weaver, "Learning to Play Chess Using Temporal Differences," *Machine Learning*, vol. 40, no. 3, pp. 243–263, 2000.

[14] C. J. C. H. Watkins and P. Dayan, "Q-Learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[15] S. M. Lucas, "Learning to Play Othello with N-tuple Systems," *Australian Journal of Intelligent Information Processing Systems, Special Issue on Game Technology*, vol. 9, no. 4, pp. 01–20, 2007.

[16] M. Buro, "From Simple Features to Sophisticated Evaluation Functions," in *Proceedings of the First International Conference on Computers and Games*. London, UK: Springer-Verlag, 1999, pp. 126–145.

[17] E. P. Manning, "Using Resource-Limited Nash Memory to Improve an Othello Evaluation Function," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 1, pp. 40–53, 2010.

[18] K. Krawiec and M. G. Szubert, "Learning N-tuple Networks for Othello by Coevolutionary Gradient Search," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '11. New York, NY, USA: ACM, 2011, pp. 355–362.

[19] M. Thill, P. Koch, and W. Konen, "Reinforcement Learning with N-tuples on the Game Connect-4," in *Proceedings of the 12th International Conference on Parallel Problem Solving from Nature - Volume Part I*, ser. PPSN'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 184–194.