Off-policy and on-policy reinforcement learning with the Tsetlin machine

Saeed Rahimi Gorji[1] · Ole-Christoffer Granmo[1]

Accepted: 27 October 2022 / Published online: 3 February 2023

Abstract

The Tsetlin Machine is a recent supervised learning algorithm that has obtained competitive accuracy- and resource usage results across several benchmarks. It has been used for convolution, classification, and regression, producing interpretable rules in propositional logic. In this paper, we introduce the first framework for reinforcement learning based on the Tsetlin Machine. Our framework integrates the value iteration algorithm with the regression Tsetlin Machine as the value function approximator. To obtain accurate off-policy state-value estimation, we propose a modified Tsetlin Machine feedback mechanism that adapts to the dynamic nature of value iteration. In particular, we show that the Tsetlin Machine is able to unlearn and recover from the misleading experiences that often occur at the beginning of training. A key challenge that we address is mapping the intrinsically continuous nature of state-value learning to the propositional Tsetlin Machine architecture, leveraging probabilistic updates. While accurate off-policy, this mechanism learns significantly slower than neural networks on-policy. However, by introducing multi-step temporal-difference learning in combination with high-frequency propositional logic patterns, we are able to close the performance gap. Several gridworld instances document that our framework can outperform comparable neural network models, despite being based on simple one-level AND-rules in propositional logic. Finally, we propose how the class of models learnt by our Tsetlin Machine for the gridworld problem can be translated into a more understandable graph structure. The graph structure captures the state-value function approximation and the corresponding policy found by the Tsetlin Machine.

Keywords Tsetlin machine · Explainable machine learning · Learning automata · Reinforcement learning · Temporal difference learning · SARSA

1 Introduction

The Tsetlin Machine (TM) is a novel supervised learning algorithm that combines learning automata and propositional logic to describe frequent data patterns [10]. A TM takes a feature vector of propositional values as input, which it maps to a target output using conjunctive clauses. This representation allows for capturing non-linear patterns, leveraging Disjunctive Normal Form (DNF).

The relation to DNF makes the TM suited for interpretation. As articulated by Valiant [25]: *“Such expressions appear particularly easy for humans to comprehend. Hence we expect that any practical learning system would have to allow for them”*. A TM model differs from a DNF formula, however, in that the disjunction operator is replaced by a summation operator. Thus, a TM produces an ensemble of clauses, which provides a multi-valued measure of output confidence. The resulting fine-grained output guides the learning process more robustly than simply employing the propositional output of a DNF formula. Yet, interpretability is maintained because one can interpret the clauses individually due to the linear decomposition of a TM model. Indeed, a TM produces a formula in plain DNF for certain hyper-parameter settings.

Several researchers have lately explored various TM-based natural language processing models, including text

This article belongs to the Topical Collection: *Emerging Topics in Artificial Intelligence Selected from IEA/AIE2021*
Guest Editors: Ali Selamat and Jerry Chun-Wei Lin

✉ Saeed Rahimi Gorji
saeed.r.gorji@uia.no

Ole-Christoffer Granmo
ole.granmo@uia.no

1 Centre for Artificial Intelligence Research, University of Agder, Grimstad, Norway

classification [4, 21], novelty detection [5], semantic relation analysis [22], and aspect-based sentiment analysis [27], using conjunctive clauses to capture textual patterns. Other application areas are network attack detection [11], keyword spotting [12], biomedical systems design [16], and game playing [9]. Further, the vanilla TM has been significantly extended by weighted clauses [13], regression architectures [2], and the elimination of hyperparameters [15]. Scalable general-purpose architectures have been proposed for CPU[1] [14] and GPU[2] [1], while special-purpose energy-efficient hardware supports learning automata-based computations [23, 26]. Moreover, theoretical work on convergence has recently appeared [28].

Presently, neural network-based models provide state-of-the-art accuracy for many pattern recognition tasks. However, their black-box nature raises concerns, making methods for explaining black-box models a main focus of contemporary research [17, 29]. Such explanations are approximations by nature and can be arbitrarily inaccurate [19]. As an alternative to explaining black-box models, pattern recognition approaches that are inherently interpretable attract an increasing number of researchers. The TM is one such approach, distinguished by combining interpretability with competitive accuracy [3, 9, 27]. Yet, work on interpretable approaches to reinforcement learning is scarce, with the TM being fundamentally supervised. This paper proposes an approach to turning the TM into an effective reinforcement learning algorithm. Using supervised learning in the context of reinforcement learning has a long history [6–8, 18]. The authors of the latter papers tried to remedy slow or lacking reinforcement learning convergence due to incremental and online learning, leveraging supervised learning. For instance, [7] proposed a reinforcement learning algorithm where a training set is prepared in each iteration to enable supervised learning. They report faster and more reliable convergence when utilizing supervision in reinforcement learning.

Today, batch-learning methods, such as experience replay, are often used to train reinforcement learning agents. Our off-policy approach presented in this paper is similar to batch-learning in the sense that we collected training data by interacting with the environment over several episodes. After data collection, a regression TM is trained using reinforcement learning principles. In addition, we propose an on-policy approach which evolves its training data as the learning progresses. This focused learning supports solving more complicated and larger scale problems.

Paper contributions This paper proposes and explores approaches to using the TM in reinforcement learning, both off-policy and on-policy. We demonstrate the viability of using bootstrapping for TM learning, lacking a pre-labelled training set. In particular, we show that the TM is able to unlearn and recover from the misleading and inaccurate experiences that inherently occur at the beginning of bootstrap-based learning before accurate state-value estimates start to propagate throughout the state-space. A key challenge that we address is mapping the intrinsically continuous nature of reinforcement learning state-value learning to the propositional nature of the TM, leveraging probabilistic updates. On-policy, this mechanism learns significantly slower than neural networks. However, we show that multi-step temporal-difference (TD) learning closes the performance gap when combined with high-frequency propositional logic patterns. Finally, we propose how the class of models learnt by our TM for the gridworld problem can be translated into an interpretable graph structure. The graph structure captures the state value function approximation and the corresponding policy found by the TM.

The rest of the paper is organized as follows. Section 2 describes the TM, its learning process, and the regression variant that we use in this paper. Section 3.1 defines the off-policy reinforcement learning problem and the details of our proposed approach. Then we introduce on-policy learning with TM in Section 3.2, covering multi-step approaches in Section 3.3. Finally, the empirical results and their analysis are presented in Section 4, while Section 5 concludes the paper.

2 Tsetlin machine

In this section, we first describe how the TM classifies input. Then we describe how TMs learn a classification model from data [10]. After that, we explain how the TM can be modified to solve regression problems [2] and suggest how the learning speed of TMs can be controlled by manipulating the number of automata states.

2.1 Classification

A TM obtains a vector $X = (x_1, \ldots, x_o)$ of propositional features as input, to be categorized into one of two classes, $y = 0$ or $y = 1$. Together with their negated counterparts, $\bar{x}_k = \neg x_k = 1 - x_k$, the features form a literal set $L = \{x_1, \ldots, x_o, \bar{x}_1, \ldots, \bar{x}_o\}$.

A TM pattern is expressed as a conjunctive clause C_j (1), built by ANDing a subset $L_j \subseteq L$ of the literal set:

$$C_j(X) = \bigwedge_{l \in L_j} l = \prod_{l \in L_j} l. \tag{1}$$

[1] https://github.com/cair/pyTsetlinMachine
[2] https://github.com/cair/PyTsetlinMachineCUDA

E.g., the clause $C_j(X) = x_1 \wedge \bar{x}_2 = x_1\bar{x}_2$ consists of the literals $L_j = \{x_1, \bar{x}_2\}$ and outputs 1 iff $x_1 = 1$ and $x_2 = 0$.

The number of clauses is controlled by a parameter n, set by the user. The first half of the clauses are given positive polarity. The second half is given negative polarity. The clause outputs are combined into a categorization decision through summation and thresholding using the unit step function $u(v) = 1$ **if** $v \geq 0$ **else** 0:

$$\hat{y} = u\left(\sum_{j=1}^{n/2} C_j^+(X) - \sum_{j=1}^{n/2} C_j^-(X)\right). \quad (2)$$

In other words, categorization is performed based on a majority vote where the positive clauses vote for $y = 1$ and the negative for $y = 0$. The classifier $\hat{y} = u\left(x_1\bar{x}_2 + \bar{x}_1x_2 - x_1x_2 - \bar{x}_1\bar{x}_2\right)$, for instance, captures the XOR-relation (illustrated in Fig. 1).

2.2 Learning procedure

Each clause j is compiled by a dedicated team of Tsetlin Automata (TA), one TA per literal l_k in L. The TA for literal l_k has a state $a_{j,k}$, which decides whether l_k is included in clause j. Learning which literals to include is based on two types of reinforcement: Type I and Type II. Type I feedback distils frequent patterns, while Type II feedback makes the patterns more discriminative.

TMs learn on-line by handling one training example $(\mathbf{X}, y)$ at a time. In all brevity, after the forward pass through the layers shown in Fig. 1 and specified in Section 2.1, each clause is updated according to Algorithm 1. The first step is to decide whether the clause is to be updated (Lines 1-3). Here, a resource allocation mechanism stimulates to create a balanced representation of the frequent patterns. This is achieved by gradually dropping the probability of updating clauses as the voting sum v approaches a user-set target T for $y = 1$ (and $-T$ for $y = 0$) for any input $\mathbf{X}$.

As seen, if a clause is not reinforced, it does not give feedback to its TAs. These are then left unmodified. In the extreme, when the voting sum v equals or exceeds the target T (the TM has successfully recognized the input $\mathbf{X}$), no clauses are updated. They are then free to learn new patterns, naturally balancing the pattern representation resources [10].

If a clause is going to be updated, the updating is either of Type I or Type II:

Type I feedback is given to clauses with positive polarity when $y = 1$ and to clauses with negative polarity when

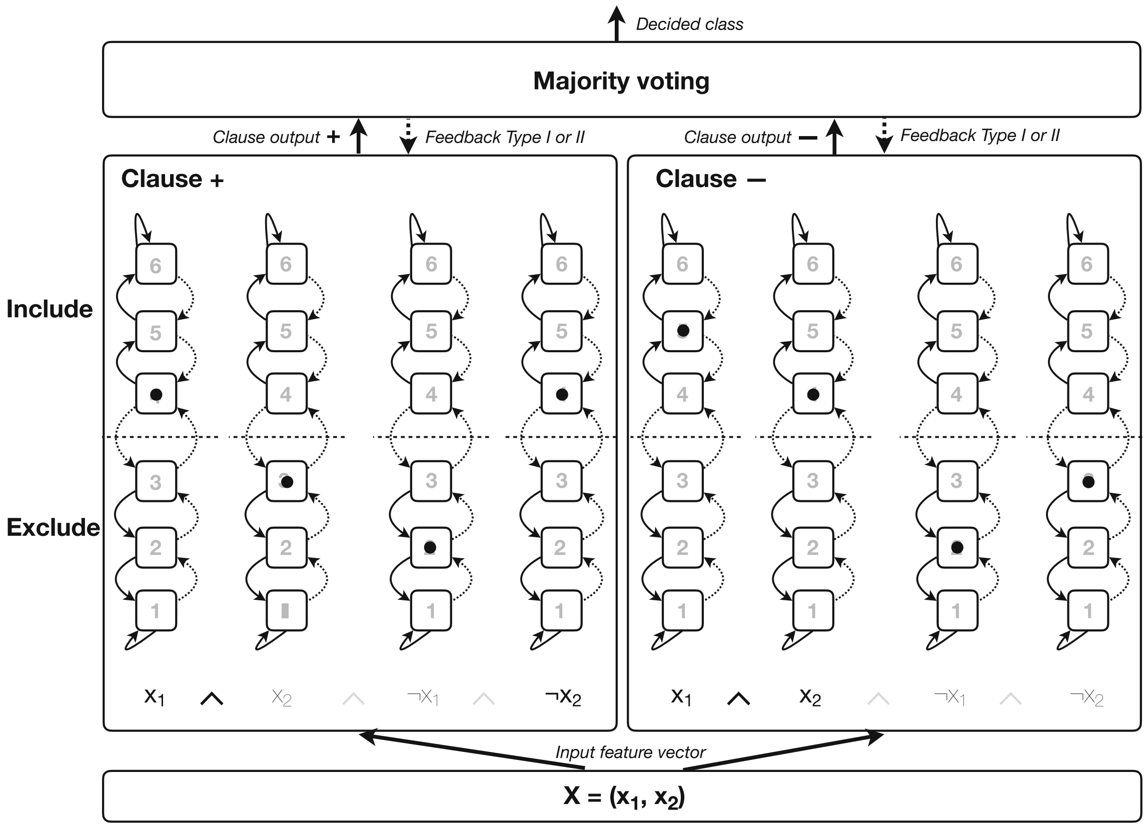


Fig. 1 The TM architecture. Each clause is constructed using Tsetlin Automata (TA) for each literal. The different TAs learn to include or exclude the different literals. The number of clauses is chosen beforehand. Here, only one positive and one negative clause are shown

```
Parameters:
    Training example $(\mathbf{X}, y)$
    Voting sum $v$
    Clause output $c_j$
    Positive polarity indicator $p_j \in \{0, 1\}$
    Voting target $threshold$ $T \in [1, \infty)$
    Pattern specificity $s \in [1.0, \infty)$
1 $v^c \leftarrow$ clip $(v, -T, T)$
2 $e \leftarrow T - (2 \cdot y_i - 1) \cdot v_i^c$
3 if $rand() \leq \frac{e}{2T}$ then
4     if $y_i$ xor $p_j$ then
5         TypeIIFeedback$(\mathbf{X}, c_j)$
6     else
7         TypeIFeedback$(\mathbf{X}, c_j, s)$
8 return
```

Algorithm 1 **UpdateClause**$(\mathbf{X}, v, c_j, p_j, T, s)$.

$y = 0$ (Line 7). Each TA of the clause is then reinforced based on: (i) the clause output c_j; (ii) the action of the TA – *include* or *exclude*; and (iii) the value of the literal l_k assigned to the TA. Two rules govern Type I feedback:

- *Include* is rewarded and *exclude* is penalized with probability $\frac{s-1}{s}$ whenever $c_j = 1$ **and** $l_k = 1$. This reinforcement is strong (triggered with high probability) and makes the clause remember and refine the pattern it recognizes in $\mathbf{X}$.[3]
- *Include* is penalized and *exclude* is rewarded with probability $\frac{1}{s}$ whenever $c_j = 0$ **or** $l_k = 0$. This reinforcement is weak (triggered with low probability) and coarsens infrequent patterns, making them frequent.

Above, hyper-parameter s controls pattern frequency.

Type II feedback is given to clauses with positive polarity when $y = 0$ and to clauses with negative polarity when $y = 1$ (Line 5). It penalizes *exclude* with probability 1 **if** $c_j = 1$ **and** $l_k = 0$. If this happens, the corresponding TA state $a_{j,k}$ is increased by 1. Thus, this feedback introduces literals for discriminating between $y = 0$ and $y = 1$.

2.3 Regression

The regression TM (RTM) predicts a continuous output value based on propositional logic expressions [2]. Just like for the TM, the input to an RTM is a vector $\mathbf{X}$ of o propositional features x_k, $\mathbf{X} \in \{0, 1\}^o$. Again, the features are expanded with their negations $\bar{x}_k = 1 - x_k$ producing a literal vector: $\mathbf{L} = (x_1, \ldots, x_o, \bar{x}_1, \ldots, \bar{x}_o) = (l_1, \ldots, l_{2o})$. As opposed to a plain TM, RTM outputs a continuous value, normalized to the domain $y \in [0, 1]$.

[3]Note that the probability $\frac{s-1}{s}$ is replaced by 1 when boosting true positives.

Prediction The regression function of an RTM is a linear summation of products, where the products are built from the literals:

$$y = \frac{1}{T} \sum_{j=1}^{n} \prod_{l \in L_j} l. \tag{3}$$

In (3) above, the index j refers to one particular product of literals, defined by the subset L_j of literal indexes. If we e.g. have two propositional variables x_1 and x_2, the literal index sets $L_1 = \{1, 4\}$ and $L_2 = \{2, 3\}$ define the function: $y = \frac{1}{T}(x_1\bar{x}_2 + \bar{x}_1x_2)$. The user set parameter T decides the resolution of the regression function. Notice that each product in the summation either evaluates to 0 or 1. This means that a larger T requires more literal products to reach a particular value y. Thus, increasing T makes the regression function increasingly fine-grained. Finally, note that the number of conjunctive clauses n in the regression function also is a user set parameter, which decides the expression power of the RTM.

Learning The RTM employs two kinds of feedback, Type I and Type II, further defined below. Type I feedback triggers TA state changes that eventually make a clause output 1 for the given training example $\mathbf{X}$. Conversely, Type II feedback triggers state changes that eventually make the clause output 0. Thus, overall, regression error can be systematically reduced by carefully distributing Type I and Type II feedback:

$$Feedback = \begin{cases} \text{Type I}, & \text{if} \quad y < \hat{y}_i, \\ \text{Type II}, & \text{if} \quad y > \hat{y}_i. \end{cases} \tag{4}$$

In effect, the number of clauses that evaluates to 1 is increased when the predicted output is less than the target output ($y < \hat{y}_i$) by providing Type I feedback. Type II feedback, on the other hand, is applied to decrease the number of clauses that evaluates to 1 when the predicted output is higher than the target output ($y > \hat{y}_i$).

2.4 Controlling adaptation by the number of Tsetlin automata states

The vanilla TM uses 8 bits of memory per automaton (depth of 128 states for each of the two actions), and it needs many training examples to learn or unlearn a pattern, which is appropriate in supervised learning with the goal of finding deep patterns in the data. In the context of reinforcement learning, however, the unlearning ability proves crucial because the algorithm must gain experience and forget previous misleading information. Indeed, the TD method relies on learning an approximation of the value function in order to find the optimal policy. Since the value function is initialized arbitrarily, the initial values do not provide any meaningful information except at the terminal states, which correspond to the value zero. As the learning progresses, however, the information about the value function gradually propagates throughout the state space, which improves the quality of the training examples. In other words, TD learning produces numerous small updates corresponding to their current value estimates as the only available source of information. As the value estimates improve, so does the quality of the updates. As such, potentially inaccurate or misleading updates are an inherent part of the learning process in the search for a solution that cannot be avoided. Thus, higher *state-bits* values such as 7 or 8 bits result in poor performance and delayed convergence by several orders of magnitude. Therefore, we addressed this issue by configuring shallower memories (3, 4, and 5 bits). Lower *state-bits* values correspond to faster learning and unlearning abilities at the potential expense of lower stability due to higher degrees of exploration. We investigated the aforementioned trade-off, using multiple *state-bits* values in our experiments. In the following, we will refer to these TMs with a different number of state-bits as TM3, TM4 and TM5, corresponding to *state-bits* values of 3, 4 and 5, respectively.

3 Reinforcement learning with the Tsetlin machine

In this section, we introduce novel TM-based approaches to reinforcement learning, going beyond the supervised TM learning paradigm. By proposing a scheme for learning from reinforcement, we address several challenges.

Challenge 1 - bootstrap-based off-policy TM learning We first explore the feasibility of using the TM as the function approximator in an off-policy RL algorithm, by employing bootstrap-based training. Our goal is to establish TM as an approach to learning an accurate representation of the state-value function for the whole state space of an RL problem. To that end, we use the value iteration algorithm in an off-policy manner.

Challenge 2 - adapting SARSA and multi-step TD for on-policy TM learning We next improve and scale up our approach. This includes adapting the SARSA temporal difference algorithm for TM learning, which is a superior on-policy algorithm compared to the above value iteration approach. Moreover, we further improve the algorithm by means of multi-step TD. As explored in Section 5, adopting these mechanisms boosts performance by several folds. Furthermore, we compare the performance and stability against a two-layer neural network as a benchmark.

Off-policy vs. on-policy TM learning The key difference between the above two approaches is that in the first one (off-policy learning), we focus on learning an accurate estimation of the value function as the primary objective, which results in learning the optimal policy as a byproduct. In the second approach (on-policy learning), on the other hand, the algorithm simply searches for the optimal policy, and the value function only plays an instrumental role. In other words, the TM learns the value function locally, on a per-need basis. As the policy evolves over time, it unlearns the irrelevant values and learns values associated with the currently relevant states. Thus, the second approach has more potential for scalability and handling large-scale problems.

Challenge 3 - booleanizing the target function A final consideration in using the RTM outside the realm of supervised learning relates to the range of values that it can learn. An RTM needs to know the smallest ($Y_{\min}$) and largest ($Y_{\max}$) output values for its learning interval $[Y_{\min}, Y_{\max}]$, which is discretized uniformly into $T + 1$ bins (where T stands for the $threshold$ value 1). Each bin is associated with the mean of its values, and together the bins form the set of all the output values that the algorithm could possibly learn. Thus, the theoretical accuracy of learning a value is within the order of magnitude of the bin size $(Y_{\max} - Y_{\min})/T$, which we refer to as the discretization gap. In supervised learning, the RTM uses the training set to infer the aforementioned bounds. For reinforcement learning, however, in the absence of an explicit training set, the RTM does not have access to any explicit upper or lower bounds beforehand. Ideally, these bounds should be learnt as a part of the learning process, but tackling such a challenge is beyond the scope of this work. Instead, we manually select effective values by analysing each problem instance.

3.1 Off-policy learning with the Tsetlin machine

To establish a foundation for further research, we first propose a solution for the supervised TM algorithm to learn a state-value function and solve simple gridworld problems. To that end, we employ the RTM, which reportedly can learn an accurate function approximation in a supervised learning manner [2]. The challenge, however, comes from the lack of a training set in an RL setting. That is, we need to train the RTM through bootstrapping based on its own experiences. These are partially learned and, therefore, potentially inaccurate state-values. Furthermore, we propose a method to cope with the exploration-exploitation dilemma of reinforcement learning.

Our approach combines the RTM and a standard reinforcement learning algorithm (value iteration) to solve deterministic instances of the gridworld problem. The approach employs a model to infer the next state given (1) the current state and (2) an action (e.g., up, down, left, right). We train the RTM to learn the state-value function for the whole state space by interacting with the environment. As a result, the optimal path is found as a byproduct. Although in general, learning a value function is just a means to an end in solving an RL problem, here we focus primarily on learning the value function accurately by exploring and addressing potential issues related to training the RTM through bootstrapping.

In the design of our reinforcement learning algorithm with the RTM, we utilize supervised RTM learning as the building block. Nonetheless, in the absence of a training set, we generate training examples through interaction with the environment and bootstrapping on the RTM's own predictions. More specifically, each iteration consists of the following steps:

1. Generating an episode (a path from the starting point to the terminal), using the ϵ-greedy policy with $\epsilon = 1$ (taking random actions at each state) as the behaviour policy.
2. Generating training examples along the episode for each state visited. The corresponding target values are calculated based on the updating formula shown in (5), which assumes taking the best action and bootstraps on the next state.
3. Training the RTM on the resulting training examples from the previous step (cf. Algorithm 2 and Algorithm 3)

```
Parameters:
    State-value function $V(.)$
    Learning rate $\alpha$
    Threshold value $T$
    The current state $S$
    $\epsilon = 1$
1  Initialize $V(S)$, for all states $S$ arbitrarily except that
   $V(terminal) = 0$
2  $episode \leftarrow 1$
3  while $episode \leq max$ do
4      Generate an episode following $\pi_{\epsilon-greedy}$:
       $S_0, A_0, S_1, A_1, ..., S_{T-1}, A_{T-1}, S_T = terminal$
5      $step \leftarrow T - 1$
6      while $step \geq 0$ do
7          $V(S_{step}) \leftarrow$
           $V(S_{step}) + \alpha \cdot \max_a(R_{S,a} + V(S'_{S,a}) - V(S_{step}))$
8          $step \leftarrow step - 1$
9      $episode \leftarrow episode + 1$
10 return $V(.)$
```

Algorithm 2 Modified Value Iteration (off-policy).

```
Parameters:
    State-value function $V(.)$
    Learning rate $\alpha$
    Threshold value $T$
    The current state $S$
1  $gap \leftarrow (Y_{max} - Y_{min})/T$
2  $target \leftarrow \max_a(R_{S,a} + V(S'_{S,a}))$
3  if $|target - V(S)| \cdot \alpha \geq gap$ then
4      RTMTrain($\mathbf{X}_S$, $V(S) + |target - V(S)| \cdot \alpha$)
5  else
6      if $rand() < |target - V(S)| \cdot \alpha / gap$ then
7          RTMTrain($\mathbf{X}_S$,
           $V(S) + Sgn(target - V(S)) \cdot gap$)
8  return
```

Algorithm 3 RTM Training through Bootstrapping.

Equation (5) shows the updating formula for the value iteration algorithm. Here, α, s and s', stand for the learning rate, the current state, and the next state after taking action a, respectively. Moreover, $R_{s,a}$ represents the reward received, resulting from taking action a in state s, which is equal to -1 for all states and actions.

$$V(s) = V(s) + \alpha \cdot \max_a(R_{s,a} + V(s'_{s,a}) - V(s)) \tag{5}$$

Algorithm 2 shows a modified value iteration algorithm [24] where the updates take place asynchronously along an episode. As a consequence of random elements in the behavior policy here, the value function for all the states has a non-zero probability of being updated during each iteration. Thus, the RTM has the potential of learning the state value function for the whole state space. Ultimately, upon the convergence of the value function approximation $V(.)$, the target policy is learned implicitly by considering the best action at each state which corresponds to the maximum value approximation.

Algorithm 3 shows how the value function updates from Algorithm 2 are applied to the RTM for a state S with the corresponding feature vector X_S. The slight difference between the two algorithms relates to the discrete nature of the RTM in learning values. More specifically, when the target value for a state S (*target* variable at Line 2 in Algorithm 3) is far from the current value estimation $V(S)$, the discretization gap, denoted by the *gap* variable, does not affect the update and RTM will be updated normally (line 4 in Algorithm 3). When the target value is too close to the current value estimation, however, the new value falls into the (discretization) gap, and a normal update does not train the RTM. To counter this issue, we consider a minimum viable-sized update to take place with the probability proportional to the size of the actual update that was supposed to be applied, but was too small to have any effect (Lines 6 and 7 in Algorithm 3). Consequently, the expected change in the corresponding value of a given state over multiple iterations and updates tends to the size of the actual updates suggested by Algorithm 2.

3.2 On-policy learning with the Tsetlin machine

Building upon the off-policy foundation from the previous section, here we propose an on-policy temporal difference approach in learning the optimal policy. More precisely, we implemented the SARSA algorithm [20] alongside the RTM as the function approximator to learn the action-value function and the corresponding optimal policy. The SARSA algorithm is an on-policy TD control method that in its simplest form has an updating formula shown in (6):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot [R_{t+1} + Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad (6)$$

Above, t is the time step during the current episode, and S_t, A_t and R_{t+1} are the state, the chosen action and the associated reward at time t, respectively. Frequent application of this updating formula results in a better approximation of the true action-value function, which in turn improves the current policy. Thus, in each epoch, first the algorithm generates an episode following the ϵ-greedy policy as the behavior policy. Next, it prepares a training set according to the generated episode by bootstrapping on the state-action pairs visited along the episode, and finally, it trains the RTM on the resulting training set (Algorithm 4).

```
  Parameters:
      Action value function $Q$
      Learning rate $\alpha$
      Number of steps n
1 Initialize $Q(S, A)$, for all state-action pairs arbitrarily
  except $Q(terminal, \cdot) = 0$
2 for each iteration do
3     Generate an episode $E_\pi$ following $\pi_{\epsilon-greedy}$
4         $E_\pi$ :
      $S_0, A_0, R_0, S_1, A_1, R_1, ..., S_{T-1}, A_{T-1}, R_{T-1}, S_T =$
      $terminal$
5     for each state-action pair $(S, A)$ in $E_\pi$ do
6         $Q(S, A) \leftarrow$
          $Q(S, A) + \alpha \cdot [R_{S,A} + Q(S', A') - Q(S, A)]$
7 return $Q(\cdot, \cdot)$
```

Algorithm 4 SARSA (ON-POLICY TEMPORAL DIFFERENCE).

3.3 Multi-step temporal difference learning

In the previous section, we described our RTM-based on-policy temporal difference algorithm. As we will see in Section 4.3, its empirical results suggest eventual convergence to the optimal path. Nevertheless, as the grid size increases, the convergence slows down, and the algorithm becomes inefficient. Thus, in this section we address the scalability issue by using a more general variation of TD learning, namely multi-step TD or n-step bootstrapping ((7) and (8) and Algorithm 5). The multi-step TD algorithm (Algorithm 5) differs from the TD learning we used so far (also known as 1-step TD) in that in each iteration, instead of taking one action at a time and updating the value function accordingly, it takes several consecutive actions. Processing consecutive actions yields more useful and informative updates by speeding up the flow of information in the grid. More importantly, in the multi-step updating of (7) and (8), each iteration accumulates the rewards of multiple consecutive actions. The bootstrapping is based on the resulting accumulation, replacing multiple iterations of 1-step TD algorithm.

$$G_{t:t+n} = \begin{cases} \sum_{i=t+1}^{t+n} R_i + Q(S_{t+n}, A_{t+n}), & \text{if } t+n < T \\ \sum_{i=t+1}^{T} R_i, & \text{otherwise} \end{cases} \quad (7)$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot [G_{t:t+n} - Q(S_t, A_t)] \quad (8)$$

Similar to (6) for the 1-step case, S_t, A_t and R_{t+1} denote the state, the chosen action and the associated reward, for the time step t, respectively. Moreover, $G_{t:t+n}$ indicates the new estimation of $Q(S_t, A_t)$ which sums up the accumulated reward from taking n consecutive actions and the value function approximation at the resulting next state.

```
  Parameters:
      Action value function $Q$
      Learning rate $\alpha$
      Number of steps n
1 Initialize $Q(S, A)$, for all state-action pairs and
  $Q(terminal, \cdot) = 0$
2 for each iteration do
3     Generate an episode $E_\pi$ following $\pi_{\epsilon-greedy}$
          // $E_\pi$ :
          $S_0, A_0, R_0, S_1, A_1, R_1, ..., S_{T-1}, A_{T-1}, R_{T-1}, S_T =$
          $terminal$
4     for each state-action pair $(S_t, A_t)$ in $E_\pi$ do
5         $G \leftarrow \sum_{i=t+1}^{min(t+n-1,T)} R_i$
6         if $t + n < T$ then
7             $G \leftarrow G + Q(S_{t+n}, A_{t+n})$
8         $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot [G - Q(S_t, A_t)]$
9 return $Q(\cdot, \cdot)$
```

Algorithm 5 MULTI-STEP SARSA (ON-POLICY TEMPORAL DIFFERENCE).

From a computational point of view, a comparison between the 1-step and the n-step (or multi-step) Sarsa algorithms reveals an extra loop related to estimating the accumulated future rewards (denoted by G in line 5 of Algorithm 5). Hence, compared to 1-step Sarsa, the n-step version takes at most $n-1$ extra addition operations to find the G value. In other words, the computational complexity of the multi-step algorithm increases linearly with the number of steps. Moreover, as we will see in Section 4.4, since a multi-step algorithm converges much faster than its 1-step counterpart, the overall computational cost of convergence is lower for the multi-step approach despite the added complexity.

4 Empirical results and analysis

4.1 The gridworld problem

We evaluated and compared the algorithms on instances of the gridworld problem. Gridworld is a classic reinforcement learning environment, representing a finite MDP in the shape of a rectangular grid where the cells correspond to the states of the environment (Fig. 2). At each cell or state, four actions are available, namely north, south, east, and west. These deterministically move the agent one cell in the respective direction on the grid, given the availability of the destination cell. If the destination cell is not available, however, the agent's location remains unchanged. In each episode, the agent starts from the initial state (indicated by ◯) and continues its journey by taking actions in the grid until it reaches the terminal state (indicated by $+$). Taking each action results in a reward of -1, and the goal is to maximize the cumulative reward acquired during an episode. Accordingly, the goal is to find the shortest path to the terminal. Moreover, to increase the complexity of the corresponding MDP, some cells in the grid have a darker color representing walls. These cells block movement and are unavailable to the agent.

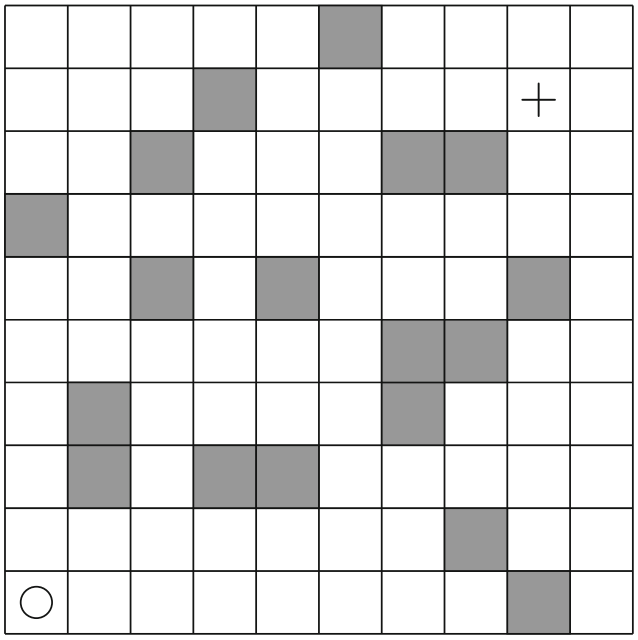


Fig. 2 The 10×10 grid with internal walls (darker tiles) for evaluating on-policy approaches

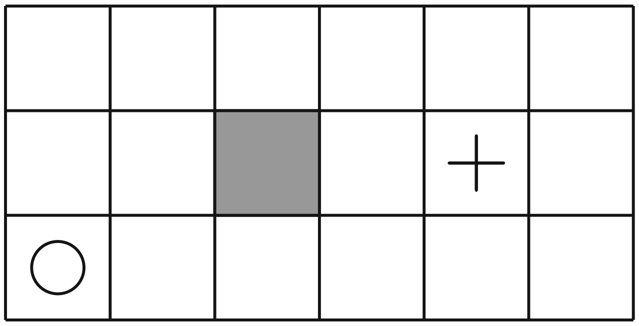


Fig. 3 The 3×6 grid (off-policy)

The RTM only accepts Boolean vectors as input, thus the first step is to encode the problem in the Boolean form. In an $m \times n$ instance of the gridworld, each state can be uniquely identified by its row and column numbers. Therefore, we encode these coordinates to an $(m+n)$-bit vector using one-hot encoding as the binary representation of a state. Moreover, we encode the chosen action in each state by using four bits corresponding to the four available actions in a similar way (one-hot encoding), which results in an $(m+n+4)$-bit feature vector representation of each state-action pair. As an example, the state-action pair $((row = 3, column = 2), action = north)$ in Fig. 2, corresponds to the feature vector resulting from concatenation of (0, 0, 1, 0, 0, 0, 0, 0, 0, 0), (0, 1, 0, 0, 0, 0, 0, 0, 0, 0) and (1, 0, 0, 0) (numbering rows from top to bottom, columns from left to right and actions in a clockwise order starting with the north). In the rest

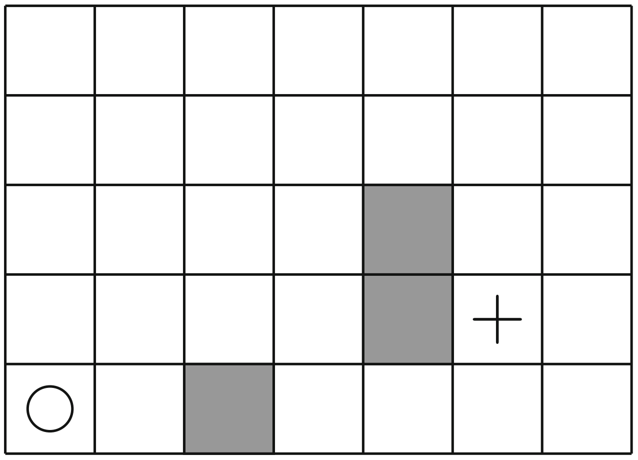


Fig. 4 The 5×7 grid (off-policy)

Table 1 Experiment configurations for off-policy learning

Grid size	Epochs (episodes)	Clauses	Threshold	$Y_{\min}$	$Y_{\max}$	Discretization gap
3 × 6	500	45	30	− 10	+ 5	0.5
		90	60			0.25
		225	150			0.1
5 × 7	500	60	40			0.5
		120	80	− 15	+ 5	0.25
		300	200			0.1

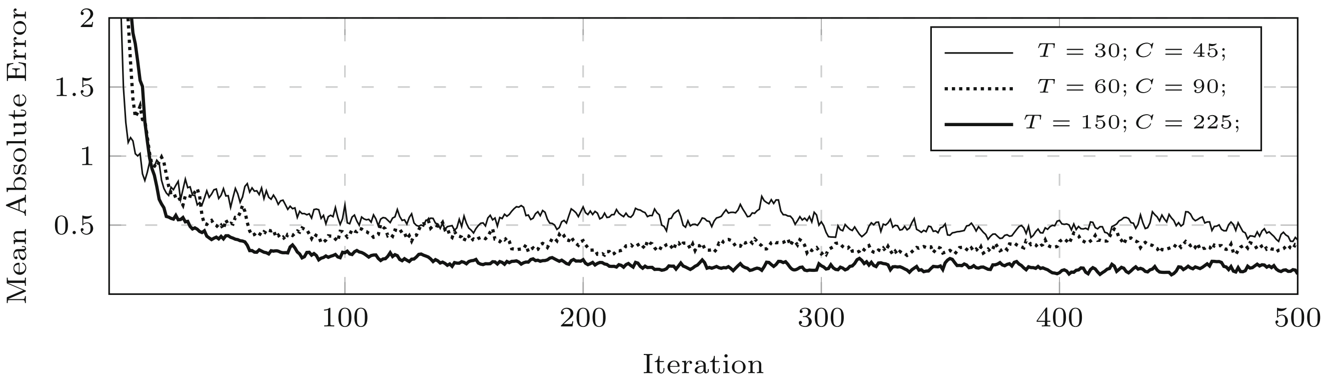


Fig. 5 Mean Absolute Error (MAE) for the 3 × 6 grid (off-policy)

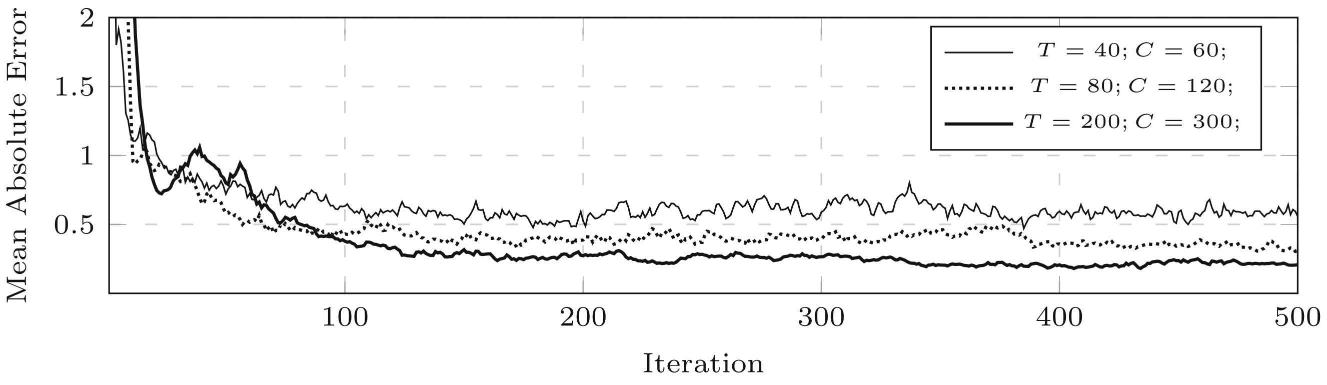


Fig. 6 Mean Absolute Error (MAE) for the 5 × 7 grid (off-policy)

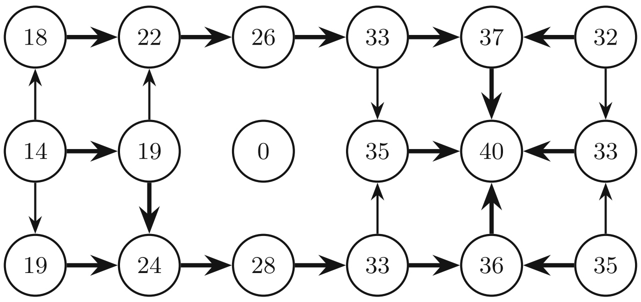


Fig. 7 Grid graph for the 3×6 grid illustrating the number of satisfied clauses at each state (off-policy)

of the paper, we use the terms “cell”, “tile” and “state” interchangeably to refer to each tile in the grid.

4.2 Off-policy learning

We used two gridworld instances of sizes 3×6 and 5×7 (see Figs. 3 and 4) to measure the performance of our approach. The two grids contain internal walls to introduce more of a challenge to the problem.

For each grid, we ran three sets of experiments with three different threshold values corresponding to discretization gaps of 0.5, 0.25 and 0.1 (Table 1). Furthermore, we used $1.5 \times \textit{threshold}$ as the number of clauses to have similar-sized TMs in terms of clause-to-threshold ratio for comparison purposes among different configurations. For each configuration, we plotted the average results over 10 independent runs. Figures 5 and 6 compare the Mean Absolute Error (MAE) of learning the state-value function for the TMs with different configurations.

After training, the RTM represents the value function approximator as a set of conjunctive clauses. We can summarize the relationship between the output clauses and

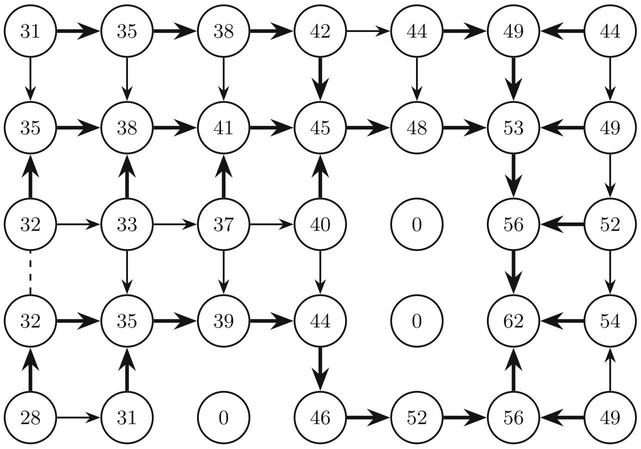


Fig. 8 Grid graph for the 5×7 grid illustrating the number of satisfied clauses at each state (off-policy)

the states of the gridworld in the form of a grid graph structure which does a better job of depicting the learned policy. In such a graph, vertices represent the states, and each vertex is labeled with the number of satisfied clauses at its corresponding state. Moreover, for each pair of neighboring states, we add a directed edge between the corresponding vertices in the direction of increasing clause satisfaction. And finally, by only considering the best action at each vertex, we obtain the RTM’s policy which consists of actions maximizing the value function approximation. As a demonstration, Figs. 7 and 8 depict the grid graphs for the cases of $gap = 0.25$, where the RTM correctly identified the best action at all states, which comprise the optimal policy (in bold edges).

4.3 On-policy learning

For each epoch, we generated one episode following the ϵ-greedy strategy. Then we built the next training set using bootstrapping examples along that episode. And finally, we trained the RTM for one pass on the training set. We continued training in this manner for 50000 epochs to investigate the convergence and robustness of the algorithms. Moreover, the results reported are the average of five independent runs to provide a more reliable view of the algorithms. In addition, considering the relatively small size of each epoch, which only included a handful of training examples, we used moving average values in the plots to clarify trends in the results.

For the RTM, we used two different thresholds, namely 200 and 400, which correspond to learning real values with the accuracy of 0.2 and 0.1, respectively. Moreover, we used three different numbers of clauses to explore the effect of increasing the size of the RTM on its performance (Table 2). Furthermore, we considered three different values for the *state-bits* parameter which is responsible for the amount of memory used by each Tsetlin automaton. On a micro level, this parameter merely controls how hard it would be for the Tsetlin automata to change their actions during training in the face of negative feedback by limiting the maximum depth of their memory. With a shallower memory, i.e. smaller *state-bits*, it would take far less negative feedback for a Tsetlin automaton to unlearn its current action and to switch to the other one. On a macro level, on the other hand, the *state-bits* parameter affects the process of unlearning a clause, which in the context of the action-value function translates into unlearning previous values and learning new ones, potentially due to finding better routes in the grid.

We compared our results with a similar approach based on multilayer perceptron (MLP) as a benchmark with a 5×5 configuration for two fully connected hidden layers and different learning rate values (α) ranging from 0.0001 up to 0.02.

Table 2 Experiment configurations for on-policy learning

Grid	Epochs (episodes)	State bits	Threshold	Clauses/Threshold	$Y_{\min}$	$Y_{\max}$
10×10	50000	3	200	5	-30	$+10$
		4	400	10		
		5		20		

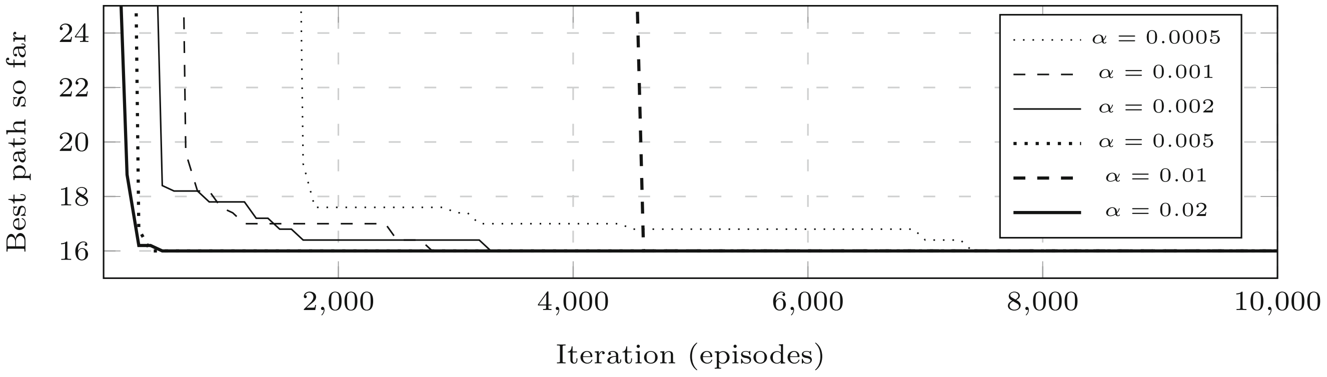


Fig. 9 1-step TD performance for MLP (on-policy)

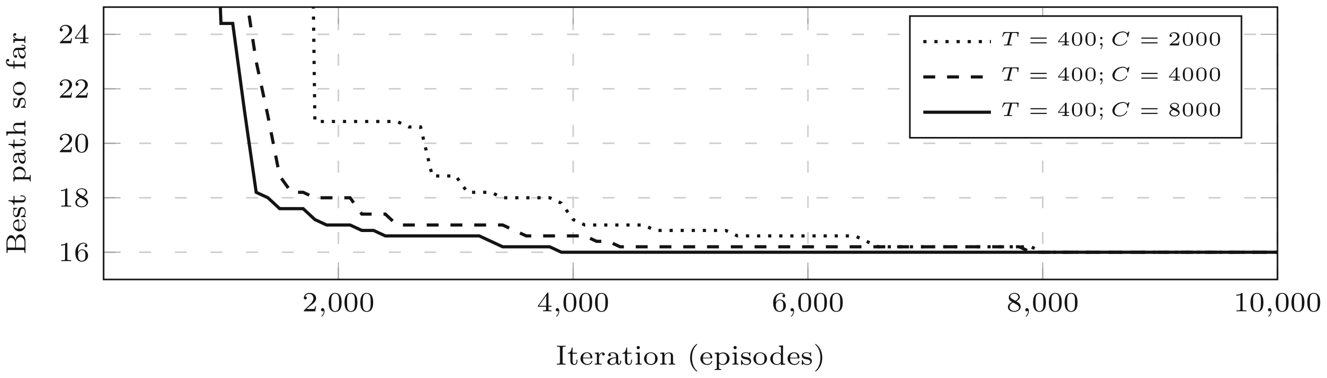


Fig. 10 1-step TD performance for TM4 with different number of clauses (on-policy)

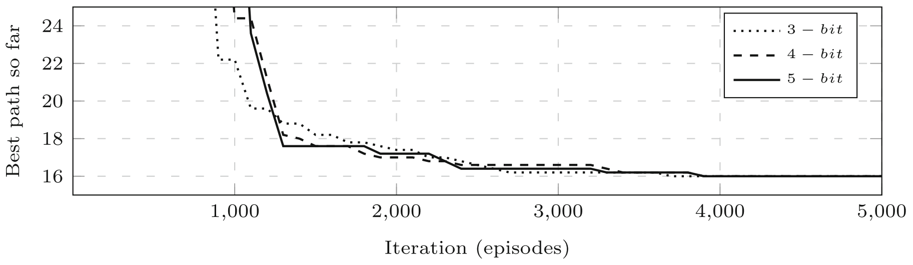


Fig. 11 1-step TD performance for TM3, TM4 and TM5 with $T = 400$ and $C = 8000$ (on-policy)

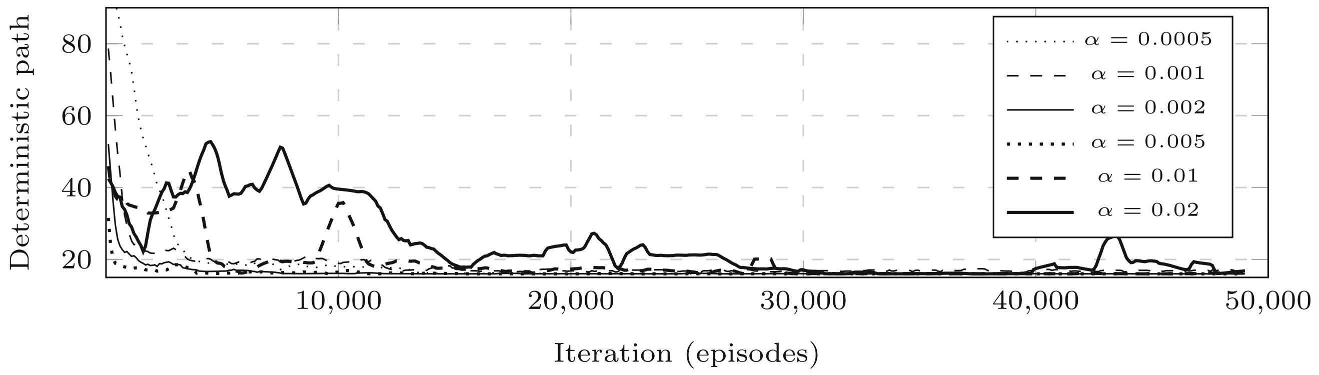


Fig. 12 Deterministic path for MLP (1-step TD) (on-policy)

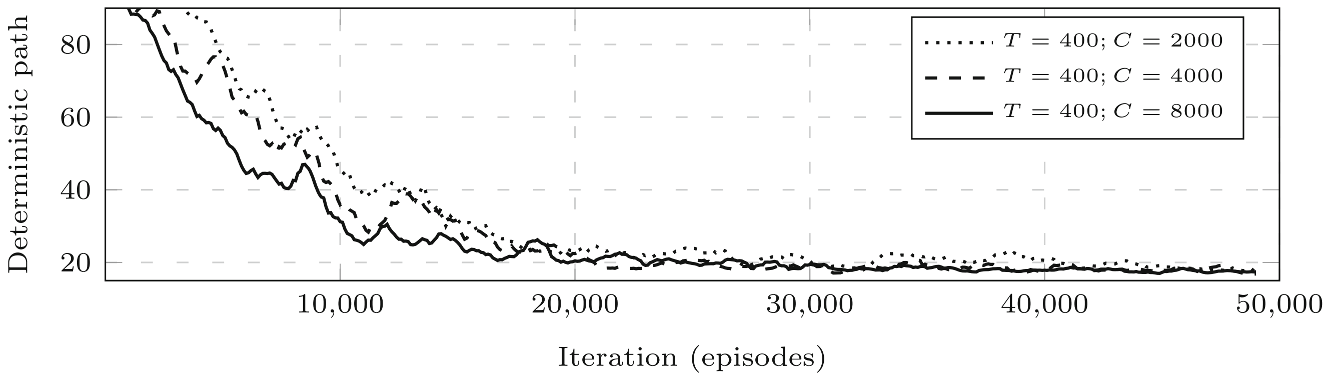


Fig. 13 Deterministic path for TM4 (1-step TD) (on-policy)

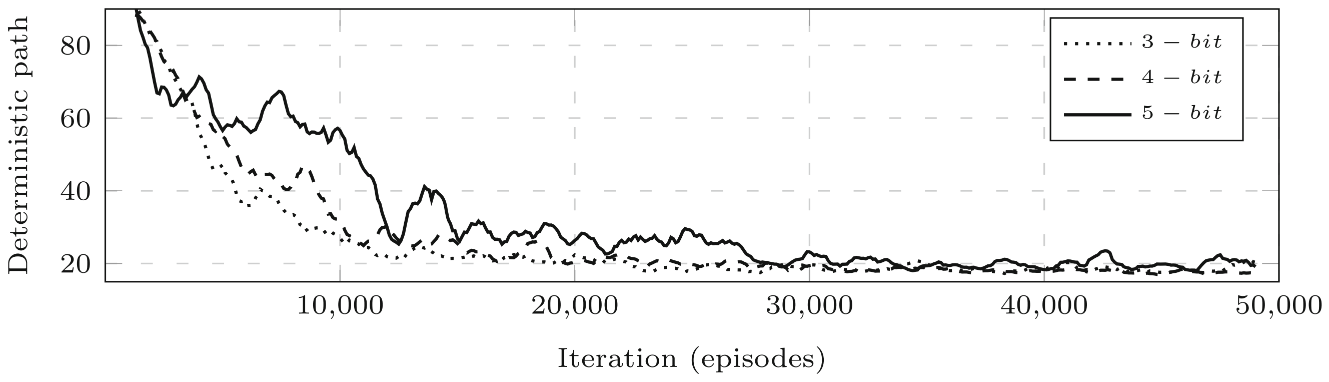


Fig. 14 Deterministic path for TM3, TM4 and TM5 with $T = 400$ and $C = 8000$ (on-policy)

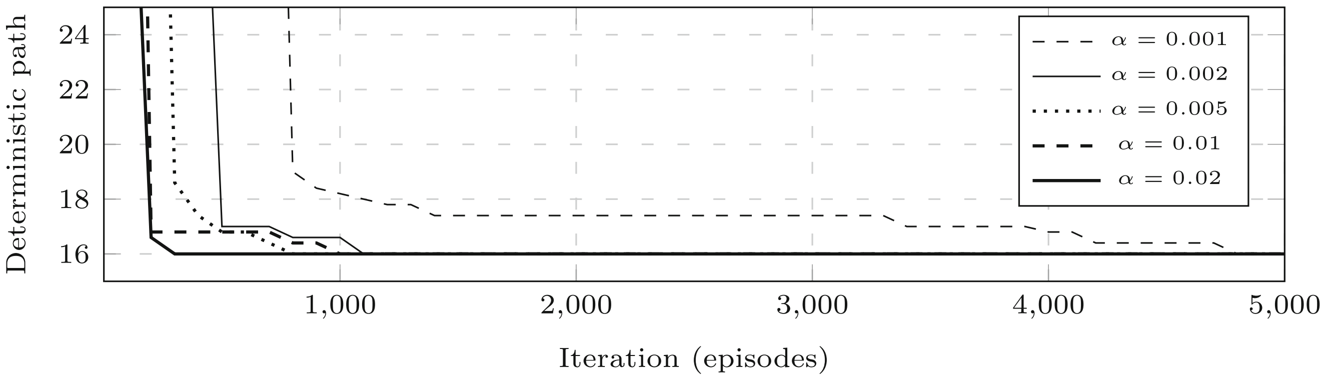


Fig. 15 2-step TD performance for MLP (on-policy)

We present two sets of empirical results, comparing the performance and the stability/convergence of the algorithms. The performance measures the average number of iterations (or episodes) needed for each algorithm to find the shortest path. The stability/convergence measure, on the other hand, is concerned with the way each algorithm behaves after finding the shortest path. Ideally, we prefer algorithms which find the shortest path in fewer episodes while being relatively stable and showing signs of converging to the shortest path as the training continues.

The first set of results (Figs. 9, 10 and 11) compares the performance or how fast the algorithms can find the shortest path. Figure 9 demonstrates that our MLP, with its relatively small size, solves the problem with ease. By increasing the learning rate (α), we observe a boost in performance and at the extreme ($\alpha = 0.02$), the MLP finds the shortest path in less than 200 epochs (episodes). But, at the same time, the overall stability decreases, as we will discuss later.

The RTM, however, falls behind in this regard, and it needs more epochs to learn the action values and, consequently, the shortest path. In Fig. 10, as expected, using a bigger RTM, i.e. more clauses, while other aspects being equal, boosts the performance. Furthermore, Fig. 11 compares the effect of different *state-bits* values on the RTM's performance. The results imply that given a sufficiently shallow memory, i.e. *state-bits* value of less than 6, the performance is the most affected by the size of the RTM.

The second set of results (Figs. 12, 13 and 14) compares the stability and convergence of the algorithms. Figure 12 indicates the instability of our benchmark MLP at extremely high learning rates, e.g. $\alpha = 0.02$, which prevents proper convergence. Thus, despite their superior performance (Fig. 9), such values are inapplicable.

For the RTM, Figs. 13 and 14 illustrate stability and convergence results. Figure 13 confirms faster convergence and improved stability for larger RTMs, which is expected because having more clauses means higher learning capacity as well as smoother transitions. Moreover, Fig. 14 suggests that given a sufficiently large RTM ($T = 400$; $C = 8000$), using a shallow memory improves stability as well as the speed of convergence. That is because unlike in supervised learning, where a shallow memory decreases stability due to frequent action changes of the Tsetlin automata, in reinforcement learning a shallow memory helps with rapid adaptation to new experiences which accelerates learning.

These results conclude that, even though the RTM can solve small-scale gridworld problems and it shows signs of

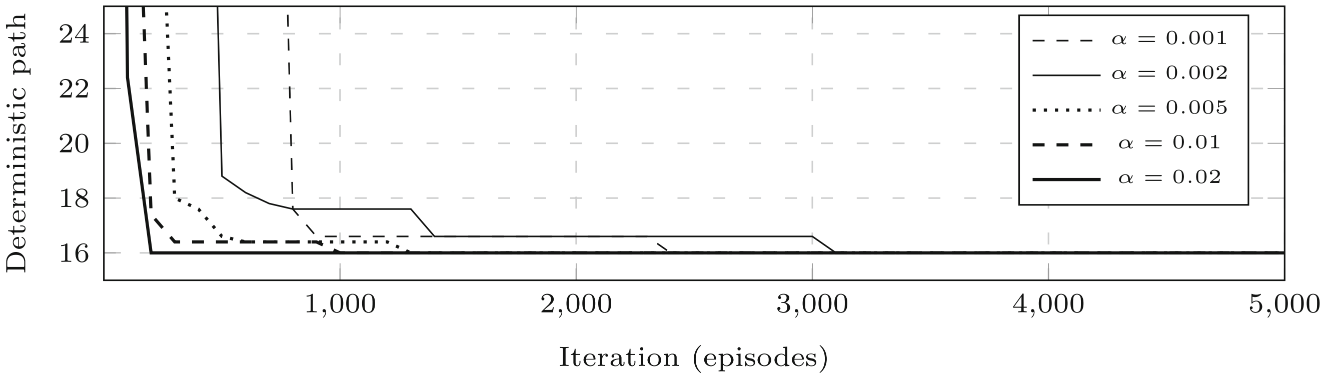


Fig. 16 5-step TD performance for MLP (on-policy)

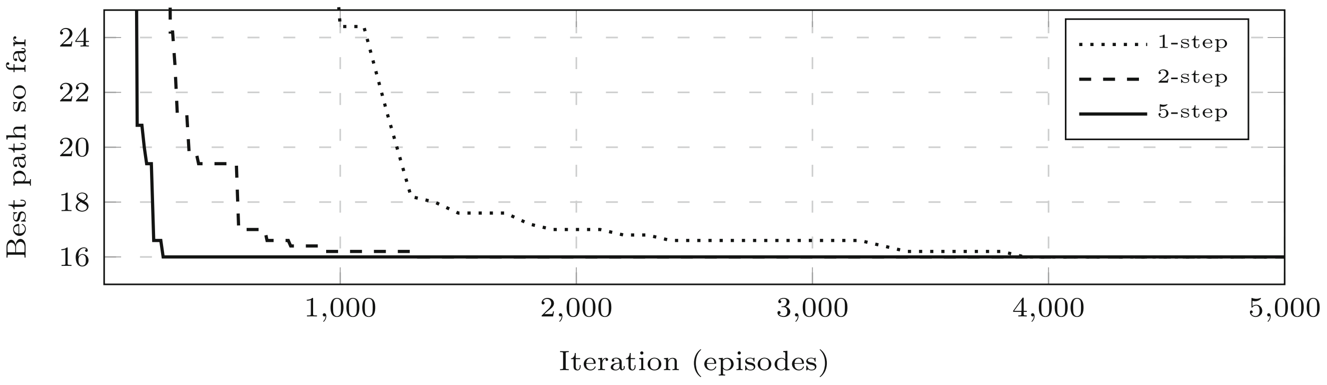


Fig. 17 Performance comparison of 1, 2 and 5-step TD for TM4 with $T = 400$ and $C = 8000$ (on-policy)

convergence and stability during learning, our benchmark MLP has a superior performance and more potential for scalability. As such, in the next section, we explore ways to improve our algorithm in terms of performance, stability and scalability.

4.4 On-policy multi-step learning

For multi-step TD, we repeated the same experiments as before. Figures 15 and 16 demonstrate the performance of 2-step and 5-step TD for the benchmark MLP. We omitted the cases with very small learning rates, namely $\alpha < 0.001$, due to their inferior performance. A comparison of these multi-step results with 1-step ones from Fig. 9 reveals a lack of improvement in performance for the multi-step variations.

Moreover, comparing results for different learning rates suggests that the learning rate itself acts as the only limiting factor on performance and that one could potentially increase it to boost the performance until the algorithm gets unstable. Subsequently, considering the fact that for $\alpha = 0.02$ or even $\alpha = 0.01$, the algorithm is unstable to some degree (as we will see later on), and that for this reason these cases should be omitted as well, one could argue that in reality the multi-step cases performs worse than the 1-step case!

In contrast, the multi-step case works really well with the RTM as it is shown in Fig. 17, and the performance improves rapidly as the number of steps increases. We see a similar effect with smaller TMs as well when using a multi-step algorithm as depicted in Figs. 18 and 19. These plots correspond to the case of a state bits value of 4 (TM4 or equivalently 4-bit TA memory). We excluded the results for TM3 and TM5 (state bits values of 3 and 5) due to similarity.

Concerning computational complexity, as we mentioned in Section 3.3, using the multi-step variation results in a linear increase in the computational cost of the algorithm. Furthermore, increasing the number 89of steps speeds up convergence to the optimal path significantly and, for instance, it takes about 4000, 1400 and 250 episodes to converge for 1-, 2- and 5-step algorithms, respectively (Fig. 17). Consequently, when put together, our empirical results indicate a reduction in the total computational cost of finding the solution by increasing the number of steps in our algorithm.

Regarding stability for multi-step variations, as we mentioned before about the MLP's performance in multi-step, when the learning rate is too low, the convergence is slow and undesirable. Similarly, for too high αs, the results get unstable and diverge, as shown in Figs. 20 and 21. Hence, these plots indicate suitable ranges of $0.0005 < \alpha <$

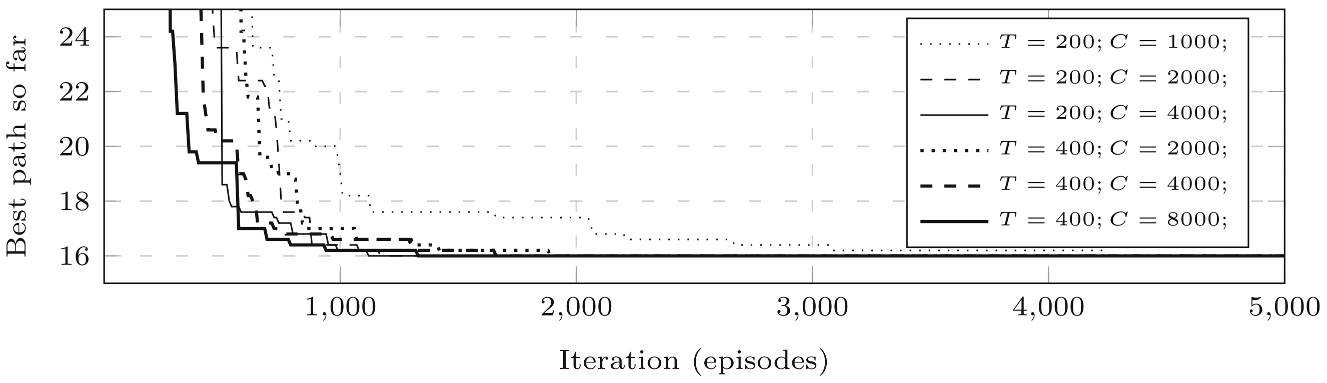


Fig. 18 2-step TD performance for TM4 with different thresholds and number of clauses (on-policy)

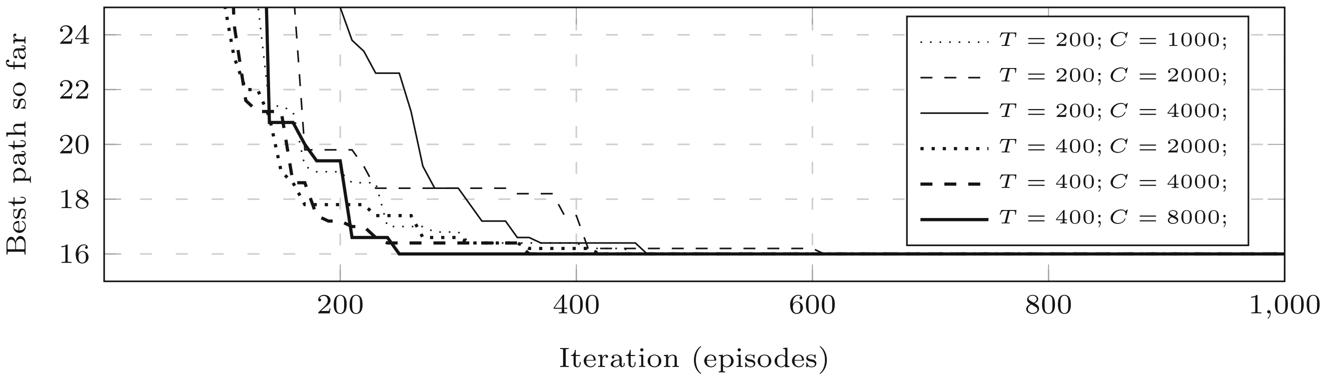


Fig. 19 5-step TD performance for TM4 with different thresholds and number of clauses (on-policy)

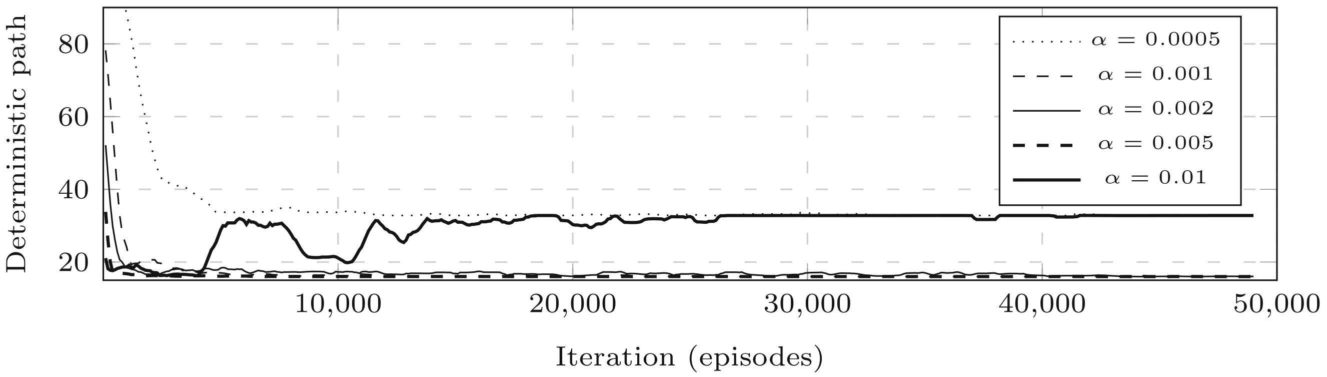


Fig. 20 Deterministic path for MLP (2-step) (on-policy)

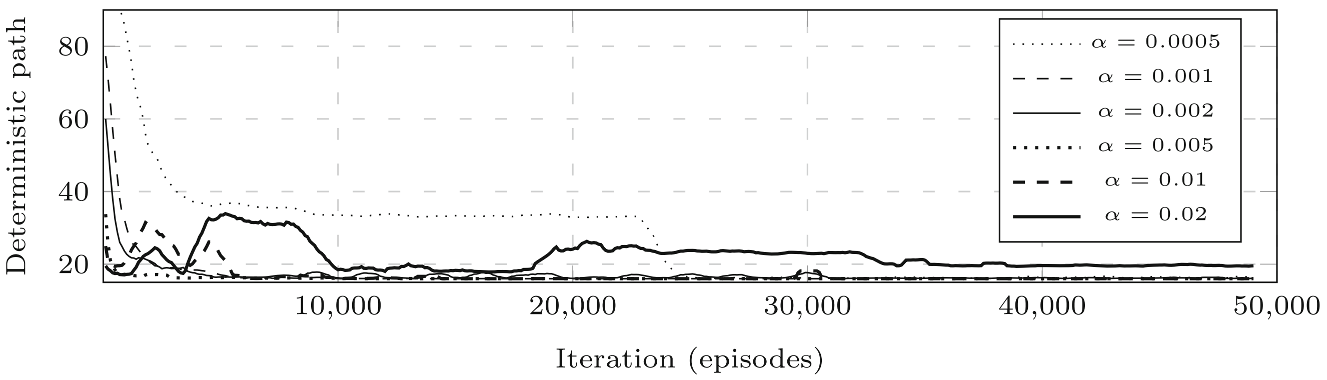


Fig. 21 Deterministic path for MLP (5-step) (on-policy)

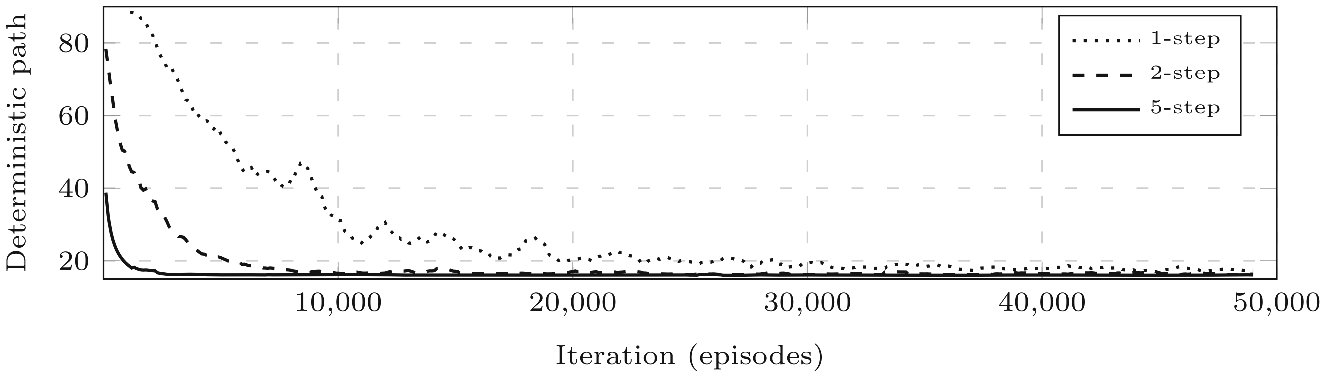


Fig. 22 Deterministic path comparison of 1, 2 and 5-step TD for TM4 with $T = 400$ and $C = 8000$ (on-policy)

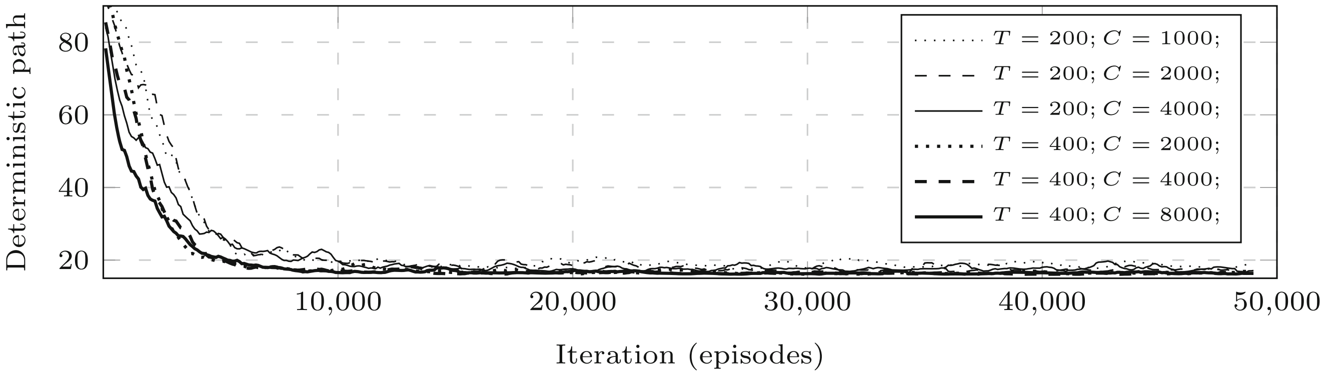


Fig. 23 Deterministic path for TM4 (2-step) (on-policy)

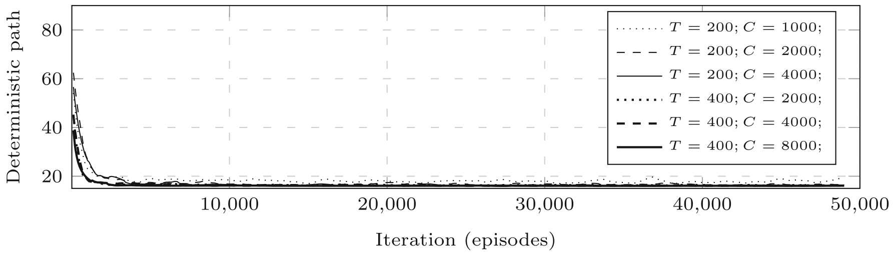


Fig. 24 Deterministic path for TM4 (5-step) (on-policy)

0.01 and $0.0005 < \alpha < 0.02$ for 2-step and 5-step TD algorithms, respectively.

Interestingly, the RTM seems as stable in multi-step learning as it was in 1-step TD, and in some cases the stability even increases, especially for bigger TMs with higher thresholds and more clauses (Fig. 22). Figures 23 and 24 illustrate multi-step results for several TMs with 4-bit memories. We omitted similar results for TM3 and TM5 (state bits values of 3 and 5).

To summarize, using the multi-step TD algorithm significantly improved the performance as well as stability of the RTM, whereas its effect on our benchmark MLP was negligible. Therefore, the RTM displayed comparable performance to our benchmark MLP in the 5-step TD results. Moreover, since increasing the number of steps in the TD algorithm could potentially improve the RTM's performance further, it could surpass such an MLP benchmark in larger instances of the problem.

It is worth emphasizing that our benchmark was a simple, small two-layered feedforward neural network, and one could argue that a more advanced architecture with more hidden layers and/or more nodes within layers configured with more suitable choices of settings will perform much better. Thus, these findings by no means paint a complete picture of neural networks' capabilities; they merely show the progress made in the direction of our research.

5 Conclusion and future work

In this paper, we introduced the TM as an RL approach, both for off- and on-policy learning. To that end, we addressed several challenges. To obtain accurate off-policy state-value estimation, we proposed a modified TM feedback mechanism that adapts to the dynamic nature of value iteration. In particular, we showed that the TM is able to unlearn and recover from the misleading experiences that often occur at the beginning of training. A key challenge that we addressed is how to map the intrinsically continuous nature of state-value learning to the propositional architecture of the Tsetlin machine, leveraging probabilistic updates. It turned out that for accurate off-policy, our mechanism learns significantly slower than neural networks on-policy. However, by introducing multi-step temporal difference learning in combination with high-frequency propositional logic patterns, we were able to close the performance gap. Our empirical results showed that using the multi-step version had a huge impact on the TM's performance, but to our surprise, the MLP did not benefit much from the multi-step approach. Consequently, although far behind in 1-step results, the TM caught up to the MLP's performance by utilizing the multi-step approach.

Our work thus forms the basis for reinforcement learning with TMs. In our further, we intend to explore how our scheme can scale up large-scale RL problems and how different classes of problems can be explained by means of the propositional TM expressions.

Funding Open access funding provided by University of Agder. Funding supporting the work conducted by the first author was partially provided by Norges Forskningsråd (Human-Chatbot Interaction Design project, grant number 270940).

Data Availability The datasets generated during and/or analysed during the current study are available from the corresponding author upon reasonable request.

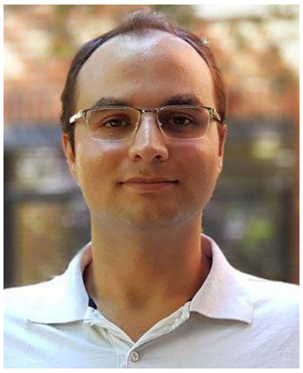
Declarations

Competing interests The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abeyrathna KD, Bhattarai B, Goodwin M, Gorji S, Granmo OC, Jiao L, Saha R, Yadav RK (2021) Massively parallel and asynchronous Tsetlin machine architecture supporting almost constant-time scaling. In: The thirty-eighth international conference on machine learning (ICML 2021). ICML
2. Abeyrathna KD, Granmo OC, Zhang X, Jiao L, Goodwin M (2019) The regression Tsetlin machine - a novel approach to interpretable non-linear regression. Phil Trans R Soc A, vol 378
3. Abeyrathna KD, Granmo OC, Goodwin M (2021) Extending the Tsetlin machine with Integer-Weighted clauses for increased interpretability. IEEE Access 9:8233–8248
4. Berge GT, Granmo OC, Tveit T, Goodwin M, Jiao L, Matheussen B (2019) Using the Tsetlin machine to learn human-interpretable rules for high-accuracy text categorization with medical applications. IEEE Access 7:115134–115146
5. Bhattarai B, Granmo OC, Jiao L (2022) Word-level human interpretable scoring mechanism for novel text detection using Tsetlin machines. Appl Intell:1–25
6. Ernst D, Geurts P, Wehenkel L (2003) Iteratively extending time horizon reinforcement learning. In: Machine learning: ECML 2003, pp 96-107. Springer Berlin Heidelberg
7. Ernst D, Geurts P, Wehenkel L (2005) Tree-based batch mode reinforcement learning. J Mach Learn Res 6:503–556

8. Ernst D, Glavic M, Geurts P, Wehenkel L (2005) Approximate value iteration in the reinforcement learning context. Appl Electr Power Syst Contr Int J Emerging Electr Power Syst, vol 3. https://doi.org/10.2202/1553-779X.1066
9. Giri C, Granmo OC, van Hoof H, Blakely CD (2022) Logic-based ai for interpretable board game winner prediction with Tsetlin machine. In: Advances in computational intelligence - IEEE world congress on computational intelligence. IEEE, WCCI 2022, Padua, Italy, 18-23 Jul 2022
10. Granmo OC (2018) The Tsetlin machine - a game theoretic bandit driven approach to optimal pattern recognition with propositional logic. arXiv:1804.01508
11. Lavrova DS, Eliseev NN (2020) Network attacks detection based on Tsetlin machine. Inf Secur Prob Comput Syst:17–23
12. Lei J, Rahman T, Shafik R, Wheeldon A, Yakovlev A, Granmo OC, Kawsar F, Mathur A (2021) Low-power audio keyword spotting using Tsetlin machines. J Low Power Electr Appl, vol 11(2). https://doi.org/10.3390/jlpea11020018, https://www.mdpi.com/2079-9268/11/2/18
13. Phoulady A, Granmo OC, Rahimi Gorji S, Phoulady HA (2020) The weighted Tsetlin machine: compressed representations with clause weighting. In: Ninth international workshop on statistical relational AI (starAI 2020)
14. Rahimi Gorji S, Granmo OC, Glimsdal S, Edwards J, Goodwin M (2020) Increasing the inference and learning speed of Tsetlin machines with clause indexing. In: Trends in artificial intelligence theory and applications. Artificial intelligence practices. Springer international publishing, cham, pp 695–708
15. Rahimi Gorji S, Granmo OC, Phoulady A, Goodwin M (2019) A Tsetlin machine with multigranular clauses. In: Lecture notes in computer science: proceedings of the thirty-ninth international conference on innovative techniques and applications of artificial intelligence (SGAI-2019). Springer international publishing, vol 11927
16. Rahman T, Shafik R, Granmo OC, Yakovlev A (2022) Resilient biomedical systems design under noise using logic-based machine learning. Frontiers Contr Eng, vol 2. https://doi.org/10.3389/fcteg.2021.778118
17. Roscher R, Bohn B, Duarte MF, Garcke J (2020) Explainable machine learning for scientific insights and discoveries. IEEE Access 8:42200–42216. https://doi.org/10.1109/ACCESS.2020.2976199
18. Rosenstein M, Barto A (2002) Supervised learning combined with an actor-critic architecture title2: Tech rep USA
19. Rudin C (2019) Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. Nature Mach Intell 1(5):206–215. https://doi.org/10.1038/s42256-019-0048-x
20. Rummery GA, Niranjan M (1994) On-line q-learning using connectionist systems. Tech Rep
21. Saha R, Granmo OC, Goodwin M (2021) Using Tsetlin machine to discover interpretable rules in natural language processing applications. Expert Syst. https://onlinelibrary.wiley.com/doi/abs/10.1111/exsy.12873
22. Saha R, Granmo OC, Zadorozhny VI, Goodwin M (2022) A relational Tsetlin machine with applications to natural language understanding. J Intell Inf Syst:1–28
23. Shafik R, Wheeldon A, Yakovlev A (2020) Explainability and dependability analysis of learning automata based AI hardware. In: IEEE 26th international symposium on on-line testing and robust system design (IOLTS). IEEE, Naples, Italy
24. Sutton RS, Barto AG (2018) Reinforcement learning: an introduction. a bradford book, Cambridge, MA, USA
25. Valiant LG (1984) A theory of the learnable. Commun ACM 27(11):1134–1142. https://doi.org/10.1145/1968.1972
26. Wheeldon A, Shafik R, Rahman T, Lei J, Yakovlev A, Granmo OC (2020) Learning automata based energy-efficient ai hardware design for IoT. Philosophical transactions of the royal society a
27. Yadav RK, Jiao L, Granmo OC, Goodwin M (2021) Human-level interpretable learning for aspect-based sentiment analysis. In: Proceedings of AAAI, Vancouver, Canada. AAAI
28. Zhang X, Jiao L, Granmo OC, Goodwin M (2021) On the convergence of Tsetlin machines for the IDENTITY- and NOT operators. IEEE Trans Pattern Anal Mach Intell
29. Zhang Y, Tiňo P, Leonardis A, Tang K (2021) A survey on neural network interpretability. IEEE Trans Emerging Topics Comput Intell 5(5):726–742. https://doi.org/10.1109/TETCI.2021.3100641



Saeed Rahimi Gorji is a PhD research fellow at the Centre for Artificial Intelligence Research (CAIR), University of Agder, Norway. He obtained his master's degree in Computer Science from Sharif University of Technology, Iran, in 2012. His research interests include Machine Learning, Learning Automata, and Game Theory.

Prof. Ole-Christoffer Granmo is the Founding Director of the Centre for Artificial Intelligence Research (CAIR), University of Agder, Norway. He obtained his master's degree in 1999 and the PhD degree in 2004, both from the University of Oslo, Norway, and created the Tsetlin machine in 2018. Dr. Granmo has authored more than 150 refereed papers with eight paper awards within machine learning, encompassing learning automata, bandit algorithms, Tsetlin machines, Bayesian reasoning, reinforcement learning, and computational linguistics. He has further coordinated 7+ research projects and graduated 55+ master- and nine PhD students. Dr. Granmo is also a co-founder of the Norwegian Artificial Intelligence Consortium (NORA). Apart from his academic endeavours, he co-founded the company Anzyz Technologies AS.