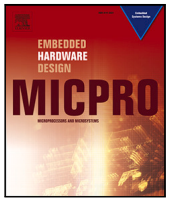
Microprocessors and Microsystems 103 (2023) 104949

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro



Convolutional Tsetlin Machine-based Training and Inference Accelerator for 2-D Pattern Classification☆,☆☆

Svein Anders Tunheim [a,*], Lei Jiao [a], Rishad Shafik [b], Alex Yakovlev [b], Ole-Christoffer Granmo [a]

[a] Centre for Artificial Intelligence Research, University of Agder, Grimstad, Norway
[b] Microsystems Group, School of Engineering, Newcastle University, Newcastle upon Tyne, UK

ARTICLE INFO

Keywords:
Machine learning
Tsetlin machine
Accelerator
Pattern classification
FPGA

ABSTRACT

The Tsetlin Machine (TM) is a machine learning algorithm based on an ensemble of Tsetlin Automata (TAs) that learns propositional logic expressions from Boolean input features. In this paper, the design and implementation of a Field Programmable Gate Array (FPGA) accelerator based on the Convolutional Tsetlin Machine (CTM) is presented. The accelerator performs classification of two pattern classes in 4×4 Boolean images with a 2×2 convolution window. Specifically, there are two separate TMs, one per class. Each TM comprises 40 propositional logic formulas, denoted as clauses, which are conjunctions of literals. Include/exclude actions from the TAs determine which literals are included in each clause. The accelerator supports full training, including random patch selection during convolution based on parallel reservoir sampling across all clauses. The design is implemented on a Xilinx Zynq XC7Z020 FPGA platform. With an operating clock speed of 40 MHz, the accelerator achieves a classification rate of 4.4 million images per second with an energy per classification of 0.6 μJ. The mean test accuracy is 99.9% when trained on the 2-dimensional Noisy XOR dataset with 40% noise in the training labels. To achieve this performance, which is on par with the original software implementation, Linear Feedback Shift Register (LFSR) random number generators of minimum 16 bits are required. The solution demonstrates the core principles of a CTM and can be scaled to operate on multi-class systems for larger images.

1. Introduction

Recently, the Tsetlin Machine (TM) was proposed as an alternative Machine Learning (ML) model [1]. The TM benefits from natural logic underpinning and low-complexity, as its foundation is propositional logic that leads to primarily Boolean operations. This operational concept of TM is hardware (HW)-friendly, and makes it highly suitable for low-power implementations [2]. The TM has shown competitive performance on several benchmarks in terms of accuracy, memory footprint and learning speed. It has been tested on tabular data [1], images [3,4], regression [5] and natural language [6,7]. In addition, the TM is highly interpretable [6–8], which makes it promising for rigorous applications, such as medical and law related solutions. The convergence properties of the TM are confirmed for basic operators in [9–11].

For embedded solutions, software (SW) implementation of ML tasks can result in low throughput and increased power consumption, due to a significant burden on the system processor. To overcome this, one can employ HW-acceleration, where a dedicated peripheral module offloads the main processor from specific demanding tasks. Such modules can be implemented by, e.g., Field Programmable Gate Arrays (FPGA), stand-alone Application Specific Integrated Circuits (ASICs) or submodules in larger System-on-Chips (SoCs).

Edge-nodes in Internet-of-Things (IoT) systems are often battery operated. If such devices are to perform ML tasks, energy-efficient solutions are needed. Frequent interaction with a cloud server is costly

☆ This work was supported by the University of Agder, Norway.

☆☆ This paper is a revised and significantly extended version of a paper presented at the First International Symposium on the Tsetlin Machine (ISTM 2022), in June 2022, in Grimstad, Norway, (Tunheim et al., 2022). The current paper presents results from a corrected and updated accelerator design with significant improvements in test accuracy. It also includes measurements of the effect of the bit length of the random number generators, and significantly more detailed explanations of the CTM-based accelerator's inference and training procedures.

* Corresponding author.
E-mail addresses: svein.a.tunheim@uia.no (S.A. Tunheim), lei.jiao@uia.no (L. Jiao), rishad.shafik@newcastle.ac.uk (R. Shafik), Alex.Yakovlev@newcastle.ac.uk (A. Yakovlev), ole.granmo@uia.no (O.-C. Granmo).

https://doi.org/10.1016/j.micpro.2023.104949
Received 18 April 2023; Received in revised form 17 July 2023; Accepted 4 October 2023
Available online 7 October 2023

in terms of processing power, data-to-decision latency and access to networks. For certain applications it could therefore also be advantageous if a node can perform online-training, i.e., update its ML model when new training samples are available. This can enable a node to adapt to changes in dynamic environment and fulfill individual learning requirements. The online learning concept can also be applied for systems utilizing federated learning. Here collaborative learning is obtained without compromising the privacy of the participants.

ML solutions at the edge are commonly implemented using Deep Neural Networks (DNNs). These systems exhibit intrinsic arithmetic as well as model complexities, making energy efficiency and online learning highly challenging. Due to the logic-based foundation of the TM, it is a promising alternative for low-power edge nodes with ML functionality, including on-device training.

Among various applications for IoT devices, 2-D pattern classification, e.g., for images, is central. For example, low-power hyperspectral sensors and cameras can be deployed for attention detection, gesture recognition and event/occupancy detection for heating/ventilation/alarm systems. Traditional solutions are mainly based on varieties of Convolutional Neural Networks (CNN). For low-power applications, Binary Neural Networks (BNNs) are gaining widespread adoption and there are numerous FPGA and ASIC implementations [12–15]. Similarly, in the TM domain, Convolutional TM (CTM) is best suited for image classification, and has obtained a peak test accuracy of 99.4% on MNIST, 96.31% on Kuzushiji-MNIST and 91.5% on Fashion-MNIST [3]. However, FPGA or ASIC implementation of CTM solutions is still an open research field.

Paper Contributions:

- This work describes a VHDL/FPGA implementation of a CTM-based accelerator for 2-D pattern classification. In more detail, it recognizes two-class patterns within 4 × 4 images. The accelerator includes complete online training of the CTM model. To our knowledge, this is the first fully functional reported HW implementation based on the CTM. The solution is designed and prepared for scaling for larger images and for multi-class classifications.
- TMs are in general highly suited for parallel operation, and we describe the degree of parallelization as the most important design trade-off regarding inference and training throughput against hardware resources.
- A key aspect of the learning phase of TMs is stochasticity. In this design, Linear Feedback Shift Registers (LFSRs) are used as random number generators. The effect of different lengths of the LFSRs on the test accuracy is evaluated.

The remainder of the paper is organized as follows. In Section 2, related studies are summarized. In Section 3, we review the general CTM operation and explain how it differs from the vanilla TM. The "2-Dimensional Noisy XOR" problem, that is studied in this work, is also presented. The accelerator architecture is described in Section 4, including important design choices and trade-offs. Measurement results of the FPGA implementation are presented in Section 5 before we conclude the work in the last section.

2. Related work

There is a wide range of HW solutions reported for image classification targeting low power edge-operation, and most of them target inference-only. As one would expect, the reported power consumption of FPGA solutions is typically several orders of magnitude higher compared to ASICs.

For edge-node operations, BNN architectures are particularly popular mainly due to their simplicity. In [12], an FPGA implementation of a BNN solution that operates at 200 MHz, achieves up to 12.3 million classifications per second on the MNIST dataset with 95.8% accuracy, and 21 906 classifications per second on the CIFAR-10 dataset with 80.1% accuracy. This is not a low-power solution as it draws about 25 W. Nevertheless, it demonstrates the capabilities of the FPGA and optimized architectures for BNN solutions.

In [16] an FPGA-based DNN inference solution, operating on the MNIST dataset, with only 16 nJ per Classification and a total power consumption of 0.16 W is reported. This is achieved with a wired-logic structure, which means that the HW is dedicated to the implemented algorithm. This is in contrast to accelerators based on a programmable architecture, which is a more desired solution in most cases.

The BNN accelerator IC module in [13] is implemented in a 22 nm Complementary Metal Oxide Semiconductor (CMOS) technology, and its energy cost per binary operation at its optimum is 21.6 fJ. It can execute the ResNet-34 BNN topology in less than 2.2 mJ per frame at 8.9 frames per second (FPS). This accelerator is included in a System-on-Chip (SoC) [14], which achieves a peak power envelope of only 674 μW and 15.4 inferences/s for the CIFAR-10 dataset.

Several analog and mixed-signal chips are reported that target ultra-low-power image classification. For example, in [17] a time-domain neural network is presented that achieves an energy efficiency of 48.2 T synaptic operations per second per watt, with an accuracy of 98.4% for MNIST. A mixed-signal BNN processor is described in [18]. The technology used is 28 nm CMOS and the power consumption is 0.9 mW. It achieves 3.8 μJ/classification on CIFAR-10 at 237 FPS.

In [15], a digital BNN test chip is reported. It is manufactured in 10 nm FinFET CMOS and achieves a peak energy efficiency of 617 TOPS/W, thus approaching the energy of analog/mixed-signal compute-in-memory solutions. It operates on the CIFAR-10 dataset and consumes 5.6 mW. The accuracy achieved is 86%.

The solutions referenced so far in this section only support training in the sense that they can be used for the classification task included in a training setup based on a system processor and external memory. One example of a solution specifically targeting training is proposed in [19]. Here an accelerator for a stochastic gradient decent based training algorithm in 16-bit fixed-point-precision numbers is described. The implementation is done both in an FPGA and in an ASIC.

From a system's perspective, one should take into account the energy and the latency for data transfer from and to the external memory, and, if required, the activation of the system's main processor. This is well reflected, e.g., in the TinyML main benchmarks [20], which are (i) *inferences per second*, (ii) *test accuracy* and (iii) *energy per classification*. For CTM-based solutions, such system-oriented performance parameters are considered favorable. This is especially the case for energy consumption, as no intermediate results need to be written to memory during inference nor training, provided the required hardware features are included. For CTM accelerators, writing to external memory is only required when a complete training has been performed and the model needs to be stored. Reading from external memory is needed for restoring a model after system wake-up, and for re-accessing training data during different training epochs.

Special encoding can compress a TM model by up to 99% without accuracy loss [21]. This implies reduced memory footprint and improved energy efficiency, and is of particular importance for ultra-low-power IoT-systems with intermittent operation.

The chip in [2] is the first TM-based ASIC reported. It supports training and inference, and implements an ultra-low-power solution for the 3-class Binary Iris data set [22]. The chip is manufactured in a 65 nm CMOS technology and is based on the *vanilla TM*.

3. Overview of CTM and problem statement

This section explains the basic operational concept of the CTM and the pattern recognition problem. The detailed *vanilla TM* operation, found in [1], is described in Appendix.

3.1. Review of the convolutional Tsetlin machine

The CTM is especially suited for 2-D image classification [3]. The examples to learn from, or to classify, are a set of images, each with dimensions $X \times Y$ and consisting of Z channels. Each channel has a pixel value represented by U bits.

In a CTM, a convolution window of size $W_X \times W_Y$ is applied, with Z channels, to generate different patches of $W_X \times W_Y \times Z \times U$ Boolean features. In addition, a clause has to be location-aware, and thermometer or one-hot encoded patch coordinates are appended to the features. In addition to improved accuracy, a main benefit of the convolutional operation, compared with the vanilla TM, is a significant reduction in the number of features that needs to be processed simultaneously.

The number of evaluations of the convolution window across the image is given by Eq. (1), where B_X and B_Y are shown in Eqs. (2) and (3) respectively. The parameters d_X and d_Y are the stride values (step sizes) of the convolution window in the X and Y directions, respectively.

$$B = B_X \times B_Y, \tag{1}$$

$$B_X = (X - W_X)/d_X + 1, \tag{2}$$

$$B_Y = (Y - W_Y)/d_Y + 1. \tag{3}$$

The number of features, N_F, applied to the CTM per patch is given by

$$N_F = W_X \times W_Y \times Z \times U + (Y - W_Y) + (X - W_X), \tag{4}$$

where $(Y - W_Y) + (X - W_X)$ represents the number of bits that encode the patch's position.

An appropriate encoding technique is applied, per channel, to the original pixels of the sample images [3]. E.g., for the MNIST dataset, there is a single channel ($Z = 1$), and each pixel value is converted to a Boolean value ($U = 1$) through simple thresholding.

The features and their negations are together denoted as the *literals*. In a CTM, the clauses can be viewed as filters, and each clause is composed by $2 \times N_F$ literals.

During inference, a CTM follows the classical TM operation for pattern recognition, as described in Appendix. However, there is a notable difference between CTM and vanilla TM. During the convolution, each clause in a CTM will output B values per image, i.e., one value per patch. After the convolution is completed, a clause in a CTM will output 1 if it has recognized a pattern at least once in any of the B patches for a given image, i.e.,

$$c_j = \bigvee_{b=0}^{B-1} c_j^b, \tag{5}$$

where c_j is the jth clause of the CTM, and c_j^b is the bth output of this clause obtained during the window sliding.

For learning, the classical TM procedure is applied with one major difference [3]: For a given clause, c_j, the CTM randomly selects a single patch among those that made this clause evaluate to 1 during the convolution. The clause is then updated according to this patch with the standard TM feedback types of Type I_a or Type II, see Appendix. If no patch made c_j evaluate to 1, Type I_b feedback will be applied, which is not dependent on the literal.

For a multi-class CTM, the Target Class is trained as a classical TM according to $y = 1$. A different class, i.e., the Negative Target Class, is randomly selected [1]. For the same training sample, the Negative Target Class is trained according to $y = 0$.

3.2. The 2-Dimensional Noisy XOR problem

The machine learning problem we adopted as a test case for the CTM accelerator is the *Two-dimensional (2D) Noisy XOR dataset* [3,23]. This contains single channel images of size 4×4, where each pixel has a Boolean value. Fig. 1 shows the patterns used for the two classes, Class 1 and Class 0. They are placed in the middle of the two upper rows. The *x*'es in the figure represent random Boolean values. The datasets generated have approximately equal numbers of Class 1 and Class 0 examples, and for each class, the different subpatterns are also represented by an equal portion. Class 1 is associated with a diagonal line, while Class 0 is associated with either a horizontal or vertical line. Thus, the dataset models a 2-dimensional version of the XOR-relation [3].

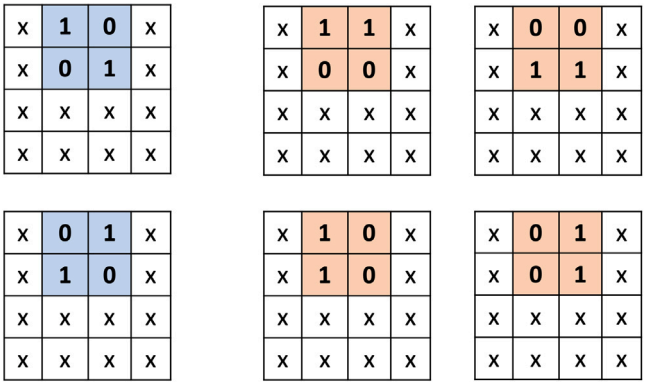


Fig. 1. Patterns representing Class 1 (blue) and Class 0 (Orange) for the 2D Noisy XOR dataset.

As the dataset includes a huge number of non-informative features (that is, the x'es in Fig. 1), it measures the CTM model's susceptibility towards the "curse of dimensionality" [3]. Furthermore, to examine the model's robustness against noise, 40% of the labels of the training data are randomly inverted. There are 2500 samples in the training dataset and 10 000 samples in the test dataset. The SW CTM implementation achieves a mean test accuracy of 99.99% for this problem. For the work described in this paper, the training dataset consists of 2500 samples while 8192 samples are used for testing.

4. Design overview

The general inference and training operations of the CTM accelerator are described in this section. In addition, we explain the window sliding, the method applied for random patch selection per clause during training, and the implementation of random number generation. High-level architectural trade-offs for latency and hardware resources are also described.

4.1. Main operation

Fig. 2 shows the block diagram of the accelerator. The module is self-contained and interfaces to the FPGA's system processor.

The 2D Noisy XOR dataset described in Section 3.2 represents a two-class classification problem. To prepare for future multi-class applications, we designed the CTM with two separate TMs, one for Class 0 (TM0) and one for Class 1 (TM1). Each of these modules contains teams of Tsetlin Automata (TAs) that form clauses, and adders that sum the clause outputs. The *Class Decision* module compares the class sums and takes the class with the largest sum as the predicted class.

We applied the hyperparameter configuration of the CTM as described in [3]. The number of clauses (per TM), m, is configured as 40, the convolution window size is 2×2, and the convolution window stride value, in both directions, is 1.

The *patch generation* module takes image samples of size 4×4 as input, and produces B patches per sample ($B = 9$ according to Eq. (1) in our case). The features per patch consists of 4 bits (from the convolution window) plus 2 bits to encode the x-position and 2 bits for the y-position. Therefore, a patch feature vector consists of 8 bits. Including

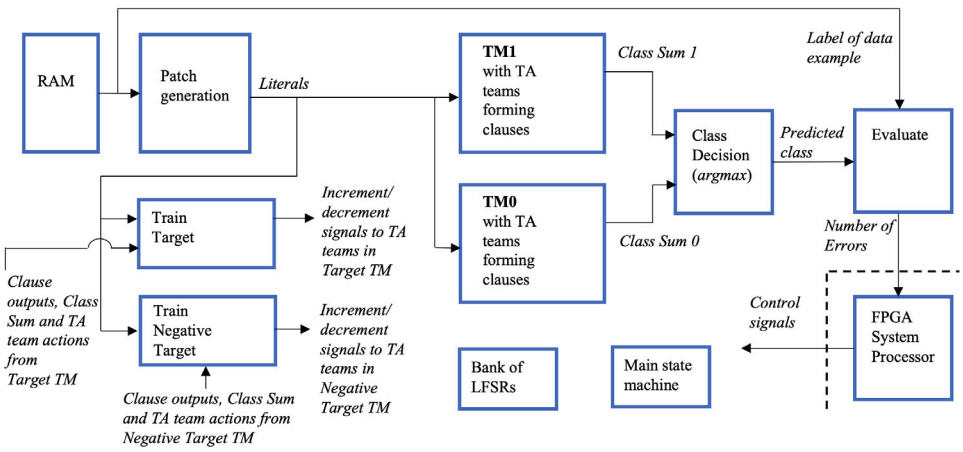


Fig. 2. Block diagram of the CTM accelerator.

the negated versions, we have a literal vector of 16 bits to be processed by the CTM accelerator per patch.

The predicted class is compared with the label of the data sample in the *Evaluate* module that keeps track of the results, i.e., the number of errors during testing.

There are two *training modules*, one for the Target Class and one for the Negative Target Class. The training modules perform the random selection of patches, per clause, to be used during updating (see Section 4.4.1) and compute the Type I_a, Type I_b or Type II feedback (see Appendix). For a given training image sample, it is the sample's label that determines whether TM0 or TM1 is to be trained as the Target Class. For this 2-class problem, the Negative Target Class is always *the other class*. A *Bank of LFSRs* is used for the generation of random numbers required during training.

For simplicity, the complete training and test datasets were initialized in the FPGA's on-chip *RAM* module via the VHDL code and the FPGA bitstream. The overall CTM operation is controlled by the *main state machine* module. It is possible to perform training or inference on just a single sample at a time.

The tasks of the *FPGA System Processor* include: (a) to program the accelerator with the number of samples to evaluate, the number of training epochs, the LFSR settings, and the selection of dataset, (b) to initiate the inference/learning session and (c) to read out the results (the number of errors) after session completion. All other functions are performed solely by the accelerator.

4.2. Inference

During inference, the data can be processed in a continuous stream without any wait states. Nine clock cycles are required per sample classification. Thus, operating the accelerator at 40 MHz, the classification rate is 4.4 million FPS.

To generate the output sum per TM, we need to add 40 numbers, corresponding to the 40 clauses. This is implemented by adders connected in a tree structure and with a 6-stage pipeline. The disadvantage with the pipeline is an additional latency of six clock periods. However, during inference, the classification rate (throughput) when operating on a stream of data samples is not affected. On the other hand, during training, see Section 4.4, this latency adds to the number of clock cycles required per sample.

To implement the function in Eq. (5), a register of width corresponding to the number of clauses ($m = 40$) is applied, i.e. *the clause output register*. During the patch generation, the contents of this register is ORed with the clause outputs for the actual patch, thus implementing a *sequential OR function*.

Clause integer weights, as described in [3,24], are not utilized in this solution due to the small image size. Thus, an alternative implementation could have been to employ a simple popcount solution – without pipelining – to find the class sums, i.e., the numbers of odd and even clauses that evaluate to 1, see Eq. (11) in Appendix. The class sum would then be the difference between these two numbers. However, to prepare for future scaled-up solutions, the design is prepared for integer weighting, and therefore adders are employed.

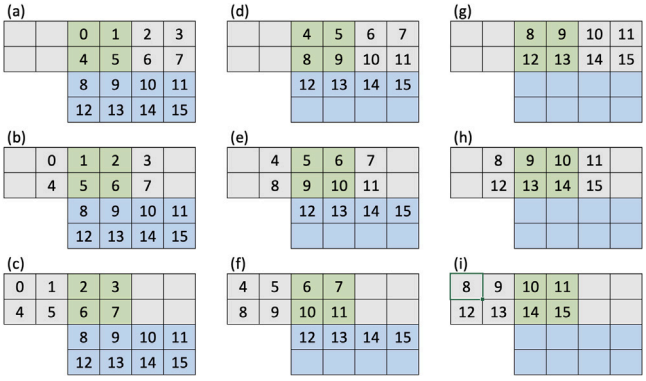


Fig. 3. Window sliding. The window position (green) is fixed, and shift operations are utilized to generate the different patches. Each square represents a D flip-flop.

The class decision is performed by comparing the class sums from TM0 and TM1, and selecting the class with the highest sum. This design is also prepared for multi-class operation, with a general *argmax* module as used in [2]. Algorithm 1 shows how inference of a single sample X is performed.

When performing inference, the modules that are only needed during training are turned off. This is implemented by setting their *register clock enable* signals low. If implemented in an ASIC, clock and power gating could be employed to reduce the power consumption further.

4.3. Window sliding

The window sliding or *patch generation* is an essential operation during the convolution, which needs to be performed efficiently. We employ simple register shift operations to achieve this. Fig. 3 shows the register structure and how the different feature patches are generated. Initially a complete picture is uploaded from RAM into four rows as shown in Fig. 3(a). Each square here represents a D flip-flop, and the green area is the window of size 2 × 2. To generate the second patch, we shift the uppermost two rows one position to the left, shown in Fig. 3(b). As can be seen, the window position is fixed, and it is the shifting of data that generates the different patches based on the content in the window area. After the last part of the uppermost two

Algorithm 1 Inference

```
procedure CLASSIFY(X)                                   ▷ X is a single image example.
    Reset clause output register.
    for i = 0 to B − 1 do                               ▷ Ref. Section 3
        In parallel during a single clock cycle:
        * Generate patch(i) of X
        * Evaluate all clauses, c_j, j ∈ {0, ..., m − 1}, for patch(i)
        * OR the clause output register content with the new clause outputs,
          and update clause output register             ▷ Ref. Eq. (5)
    end for
    Calculate class sums v(0) and v(1)                  ▷ Ref. Eq. (11)
    return argmax(v(0), v(1))                           ▷ Returns predicted class
end procedure
```

Table 1
Position encoding of the 2 × 2 convolution window within a 4 × 4 Boolean image. x with position 0 means leftmost column, while y with position 0 means uppermost row.

(x,y) position	Encoded bit pattern
(0,0)	1010
(1,0)	0110
(2,0)	0010
(0,1)	1001
(1,1)	0101
(2,1)	0001
(0,2)	1000
(1,2)	0100
(2,2)	0000

rows are positioned in the window registers, shown in Fig. 3(c), the second row is shifted up and two places to the right, and a new row is loaded into the second upper-most row, shown in Fig. 3(d). In a similar manner, we continue until the whole picture has been processed. The same principle can be applied for CTMs operating on larger multi-layer images.

A patch's position information is appended to the four bits from the convolution window. One-hot encoding, with two bits for each coordinate, is performed as shown in Table 1.

During the patch generation, the output of each clause is determined by Eq. (5) and the *clause output register* is updated for each patch evaluation.

4.4. Training

Each training module in Fig. 2 takes as inputs the following signals from each of the two separate TMs it trains: (a) the clause outputs, (b) the TM's output sum, (c) the actions of the TM's teams of TAs, and (d) the literals (that come from the randomly selected patch as described in Section 4.4.1).

During clause updating, the training module outputs – for each clause in sequence – *reward/penalty* signals to the TA teams. All 16 TAs in a clause's TA team obtain their update signals in parallel. The specific training hyperparameters are according to [3], namely, $T = 40$ and $s = 3.9$. To save HW resources and energy, fixed hyperparameters are used. This implies that all required multiplications can be performed with specific combinations of shift and addition operations.

As in [2], each TA is implemented as a binary up-down counter. We have used 8 bit counters. Thus, there are $2^8 = 256$ states in total for a TA, ranging from −128 to +127 (two's complement representation). If the TA state is 0 or higher the given literal will be included in the clause. The TA's most significant bit can therefore directly be applied as an include/exclude signal in the clause logic.

Training data can be processed in a continuous stream. The patch generation requires nine clock cycles, as for inference mode. Due to the pipelined adders in the class sum stages of the TMs, a delay of six clock cycles is necessary per sample before the output sum is ready and the clause updating can start. The clauses are then updated in sequence, one by one, and this takes 40 clock cycles. In total, the number of clock cycles required per training sample with this architecture is 55. The training procedure is carried out in parallel for the Target Class and Negative Target Class, as the patch generation is the same for both.

For random number generation required during training, a bank of LFSRs is employed, and the details are given in Section 4.4.2.

4.4.1. Reservoir sampling and patch selection

During the learning phrase, one has to randomly select a single patch, per clause, among those that made the clause evaluate to 1 during the convolution (window sliding) operation. We have implemented this by utilizing the *reservoir sampling* algorithm [25]. Specifically, version R of the algorithm is applied.

The patches that made a given clause evaluate to 1 constitute *the set* for the reservoir algorithm for the clause, and the number of these patches is n. We are interested in selecting randomly only a single sample (i.e., one patch), from the set, and the algorithm ensures that, when it has finished executing, every element of the set is selected with probability $1/n$. Only a single pass of the elements is needed with this algorithm, which fits nicely with the timing of the patch generation.

For a given clause, $c_j, j \in \{0, \ldots, m-1\}$, the input data stream to the reservoir sampling algorithm in the CTM accelerator is empty until c_j evaluates to 1. Then a register N_j is incremented, where N_j is the number of times c_j evaluates to 1 during the patch generation.

For each clause independently, a new sample, that is, patch(i), $i \in \{0, \ldots, B-1\}$, is only fed to the reservoir sampling algorithm when the given clause evaluates to 1. For each time a new sample is added, the reservoir algorithm for this clause randomly decides whether it shall replace the existing sample in the clause's *patch register*, PatchReg(j), $j \in \{0, \ldots, m-1\}$. After the patch generation is completed, PatchReg(j) will contain the randomly selected patch that will be used during updating of clause j. The maximum number of times a clause can evaluate to 1 during the patch generation is B, which is 9 in our configuration. Algorithm 2 details the reservoir sampling.

For the random patch selection, look-up tables are employed for the multiplications required. All other multiplications are implemented by specific combinations of shift and summation operations.

Independent reservoir sampling is performed in parallel for all clauses, both for the Target Class and the Negative Target Class. This is effective, but requires additional hardware resources, mainly for the registers for storing the randomly selected patches and LFSRs for generating random numbers.

4.4.2. Random number generation

Stochasticity is key for the learning phase of TMs [1]. For HW implementations, we need to generate and access random numbers with adequate stochasticity, which should also be accomplished with low power consumption.

Algorithm 2 Reservoir sampling

```
procedure RESERVOIR SAMPLING(X)                ▷ The reservoir per clause has a single element, i.e., a patch.
    Reset the count and patch registers, N(j), PatchReg(j), per clause, j ∈ {0,...,m−1}
    for i = 0 to B − 1 do                      ▷ B is the number of patches, ref. Section 3
        In parallel during a single clock cycle:
        * Generate patch(i) of X
        * Evaluate all clauses, c_j, j ∈ {0,...,m−1} for patch(i)
        if c_j = 1 then
            N(j) ← N(j) + 1
            r ← random integer in the range [1, N(j)]
            if r ≤ 1 then
                PatchReg(j) ← patch(i)
            end if
        end if
    end for
    return For each clause, c_j, j ∈ {0,...,m−1}:
            PatchReg(j) with randomly selected patches and count register N(j)
end procedure
```

Maximum-length sequences (MLS) implemented by LFSRs are suitable for generating the required random numbers [26,27], and we applied this for our design. It is desirable to have as short LFSRs as possible to limit the required HW resources (mainly D flip-flops) and the associated energy consumption. We have implemented the design with the option to choose between 6-, 7-, 8-, 10-, 12-, 14-, 16-, 18- and 24-bit LFSRs to evaluate the performance impact of the random number generators. Correct implementation of each LFSR type and their number sequence time periods are verified through VHDL simulations.

All LFSRs in the accelerator have different seeds, i.e., distinct start conditions from reset, to avoid statistical dependence between simultaneous random decisions. The only exception is the 6-bit LFSRs due to their short number sequence and therefore the limited number of possible seeds. To avoid the same start conditions per run (after reset), the LFSRs were operated for a small random time period, controlled by the system processor, before the start of the training.

4.4.3. Updating of clauses and TA teams

Algorithm 3 shows the complete training procedure, which is performed in parallel also for the Negative Target Class. After patch generation and reservoir sampling have been performed, the accelerator updates each clause, c_j, and its associated TAs, in sequence, according to Type I_a, Type I_b and Type II feedback, described in Appendix.

The details of the stochastic parallel updating of the TAs per clause are shown in Algorithm 4. This procedure is performed for each clause, c_j, $j \in \{0, m-1\}$, in sequence. Table 2 describes the various parameters for Algorithm 4.

Before learning starts, all TAs are initialized in state $N-1$ (state −1 in our design), i.e., with action *exclude* [1], see also Fig. 5 in Appendix. This implies that all clauses are empty initially. For an empty clause, the clause output is forced to 1 during learning.

4.5. Architectural trade-offs

The main design trade-off for the CTM accelerator is the classical question of parallel versus sequential operation. This determines throughput and the amount of HW resources required, and thereby also the peak power consumption.

For inference, the limiting factor of classifications per second is the patch generation. With one window register, B clock cycles are required per classification. One can parallelize this by employing several convolution windows, each operating on different parts of the image. However, the TM's clause outputs are required for all patches that are evaluated simultaneously. Therefore, one would need to include multiple copies of the TM clause logic. The TA teams' *include/exclude* signals can be reused. For example, for 4 × 4 images, an alternative implementation could be to adopt three different convolution windows, enabling a 3-fold increase in the classification rate.

Table 2
Overview of parameters in Algorithm 4.

Parameter	Description
TC	TC = 1 if Target Class, otherwise 0.
m	Number of clauses
c_j	Clause j, $j \in \{0,\dots,m-1\}$.
PatchReg(j)	Randomly selected patch, per clause, ref. Algorithm 2.
N_F	Number of features, ref. Eq. (4).
l_k	Literal k from the randomly selected Patch(j), $k \in \{0,\dots,2\times N_F-1\}$.
f	$f = 1$ if j is odd, where $j \in \{0,\dots,m-1\}$. Otherwise $f = 0$.
u_p	Stochastic signal given by Eq. (12) $u_p = 1$ if clause c_j in Target Class is to be updated.
u_n	Stochastic signal given by Eq. (13) $u_n = 1$ if clause c_j in Negative Target Class is to be updated.
g_k	Stochastic signal for literal k. $g_k = 1$ if a random number rnd_k for literal l_k is $< (s-1)/s$, else 0, ref. Table 4.
h_k	Stochastic signal for literal k. $h_k = 1$ if a random number rnd_k for literal l_k is $< 1/s$, else 0, ref. Table 4.
$TA_{j,k}$	State of TA for clause j controlling l_k.
$\alpha_{j,k}$	Action of $TA_{j,k}$. For *include* action $\alpha_{j,k} = 1$ and for *exclude* action $\alpha_{j,k} = 0$.

During training, the reservoir sampling is performed in parallel with the patch generation. This can be parallelized further in the same way as for inference. Following the reservoir sampling, the clauses are updated in sequence one by one, as shown in Algorithm 3. The motivation for this is to save LFSRs. It is possible to reduce the processing time per training sample by updating more clauses in parallel. For example, if two clauses were updated simultaneously, we would need only 20 clock cycles for the clause updating, instead of 40, reducing the total number of training clock cycles per sample from 55 to 35. The number of LFSRs in this case would be approximately equal to the number of LFSRs required for the reservoir sampling. However, increasing the parallelism for the reservoir sampling further would require more LFSRs.

The number of D flip-flops needed for the TA teams per TM, N_{TM}, is given by Eq. (6):

$$N_{TM} = 2 \times N_F \times m \times N_{TA}, \tag{6}$$

where N_F is the number of features (in our case 8, see Section 4.3), m is the number of clauses (40 per TM) and N_{TA} is the number of bits in

Algorithm 3 Training

procedure TRAIN($((X_0, Y_0), \ldots, (X_{k-1}, Y_{k-1}))$ ▷ k training examples.
  `Initialize TAs` ▷ Set all TAs to state $N-1$ (*exclude*).
  for $n = 0$ `to` $k-1$ **do**
    `Reset clause output register and patch registers.`
    for $i = 0$ `to` $B-1$ **do** ▷ Ref. Section 3
      In parallel during a single clock cycle:
      * `Generate patch(`i`) based on (`X_n`)`
      * `Evaluate all clauses,` $c_j, j \in \{0, \ldots, m-1\}$`, for patch(`$i$`)`
      * `Perform reservoir sampling for all clauses,` ▷ Ref. Algorithm 2
        `and find a random patch, per clause, and store in PatchReg(`j`).`
      * `OR the clause output register with the new clause outputs`
        `and update clause output register` ▷ Ref. Eq. (5)
    end for
    `Calculate class sums` $v(0)$ `and` $v(1)$ ▷ Ref. Eq. (11)
    for $j = 0$ `to` $m-1$ **do** ▷ Ref. Section 3
      In parallel during a single clock cycle:
      * `Randomly decide, based on` c_j `(If` $N(j)$`=0 then` $c_j = 0$`, otherwise` $c_j = 1$`),`
        `class sum,` Y_n `and` T`, whether` c_j `shall be updated` ▷ Ref. Appendix
      * `Update stochastically all TAs for` c_j ▷ Ref. Algorithm 4
    end for
  end for
  return `Updated TAs for all clauses`
end procedure

Algorithm 4 Updating of the TAs per clause - detailed operation

procedure UPDATE TAs(for clause c_j, for randomly selected patch from PatchReg(j))
  ▷ For parameter descriptions, see Table 2
  `Do in parallel for all` $TA_{j,k}, k \in \{0, \ldots, 2 \times N_F - 1\}$`,`
  `for clause` j`, during a single clock cycle:`
  if $TC \wedge u_p$ **then** ▷ Training of Target Class
    if $c_j \wedge f \wedge g_k \wedge l_k$ **then**
      $TA_j(k) \leftarrow TA_j(k) + 1$ ▷ Type Ia feedback
    else if $c_j \wedge f \wedge h_k \wedge \neg\alpha_{j,k} \wedge \neg l_k$ **then**
      $TA_j(k) \leftarrow TA_j(k) - 1$ ▷ Type Ia feedback
    else if $\neg c_j \wedge f \wedge h_k$ **then**
      $TA_j(k) \leftarrow TA_j(k) - 1$ ▷ Type Ib feedback
    else if $c_j \wedge \neg f \wedge \neg\alpha_{j,k} \wedge \neg l_k = 0$ **then**
      $TA_j(k) \leftarrow TA_j(k) + 1$ ▷ Type II feedback
    end if
  else if $\neg TC \wedge u_n$ **then** ▷ Training of Negative Target Class
    if $c_j \wedge \neg f \wedge g_k \wedge l_k$ **then** ▷ Type Ia feedback
      $TA_j(k) \leftarrow TA_j(k) + 1$
    else if $c_j \wedge \neg f \wedge h_k \wedge \neg\alpha_{j,k} \wedge \neg l_k = 0$ **then**
      $TA_j(k) \leftarrow TA_j(k) - 1$ ▷ Type Ia feedback
    else if $\neg c_j \wedge \neg f \wedge h_k$ **then**
      $TA_j(k) \leftarrow TA_j(k) - 1$ ▷ Type Ib feedback
    else if $c_j \wedge f \wedge \neg\alpha_{j,k} \wedge \neg l_k$ **then**
      $TA_j(k) \leftarrow TA_j(k) + 1$ ▷ Type II feedback
    end if
  end if
  return `Updated TAs for clause` c_j
end procedure

a TA when implemented as a counter (8 in our design). The factor of 2 is included to take into account the negated version of the features. We have two TMs (TM0 and TM1) in our design. Thus, the total number of D flip-flops required for the TA teams is 10 240.

For the patch feature registers, employed during the reservoir sampling and clause updating, the number of D flip-flops needed, N_P, is:

$$N_P = 2 \times m \times N_F, \tag{7}$$

where the factor of 2 in Eq. (7) is due to the parallel training of the Target Class and Negative Target Class. In our design N_P is configured as 640.

The number of D flip-flops required for the LFSRs is also important. For the reservoir updating, m LFSRs are employed for each of the Target Class and the Negative Target Class. As the subsequent clause updating is performed sequentially, $1 + 2 \times N_F$ LFSRs are utilized in this phase for each of the Target Class and the Negative Target Class, where the addition of 1 is due to the requirement of an additional random number generator to determine if a clause is to be updated, see Eqs. (12) and (13) in Appendix.

LFSRs applied for the reservoir sampling can be reused during the clause updating. Thus, the number of LFSRs required for our architecture is the maximum of two numbers:

$$N_{LFSRs} = \max\left\{2 \times m, 2 \times (1 + 2 \times N_F)\right\}. \tag{8}$$

For our design, the first part is the largest, and a total of 80 LFSRs are utilized. With an LFSR length of 16 bits, the number of D flip-flops needed is 1280. The total number of D flip-flops in this accelerator, from these three main contributions, is 12 160. In our design, additional D flip-flops are needed for implementing LFSRs of varying lengths. For multi-class system, the TA teams' relative contribution to the total number of D flip-flops will increase. We believe the architecture chosen for our implementation is an adequate compromise for this study, demonstrating the core CTM principles in HW, and for implementation on a low-cost FPGA.

5. Implementation results

The system configuration and the results from the experiments are detailed in this section. The CTM accelerator was designed in VHDL, and the simulations were performed with Xilinx *Vivado*. A Xilinx Zynq XC7Z020 FPGA on a Zybo Z7-20 board from Digilent was adopted for the implementation.

The accelerator occupies 44.3k Look-Up Tables (LUTs) and 25.5k D flip-flops, which corresponds to 83.3% and 24% utilization respectively of the LUTs and D flip-flops available in this FPGA. Included on the FPGA are two hardcoded ARM9 cores, and one of these processors was used to apply control signals to the accelerator and to read out results. The Xilinx *Vitis* program was applied to compile the C-programs for the system processor.

During inference, the accelerator achieves 4.4×10^6 classifications/s when operating with a clock frequency of 40 MHz. This is given directly by the design characteristics of nine clock cycles per inference, and by the solution's capability to process input data samples in a continuous stream. During training, with 55 clock cycles per sample, the accelerator processes 0.73×10^6 samples/s. Thus, a training session (run) based on 2500 samples and 250 epochs takes approximately 0.9 s. The inference and training rates described here excludes the time required for commands and reading of results by the processor. However, this is very small as all training and test data are available directly from the dataset RAMs that were initialized from the FPGA bitstream.

A simple throughput comparison between the HW accelerator and the SW implementation was performed. The server running the SW implementation was equipped with Intel Xeon Platinum 8168 CPUs operating at 2.70 GHz. With the CTM HW accelerator, we obtained a throughput increase of 13.3 times for inference and 12.1 times for training.

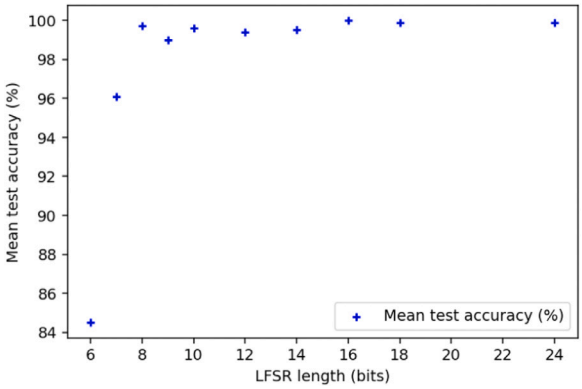


Fig. 4. Mean test accuracy of the CTM accelerator versus LFSR bit length.

The effect of the LFSR bit length on the accelerator's performance was explored. Experiments with LFSRs with bit lengths of 6, 7, 8, 10, 12, 14, 16, 18 and 24 were made. For each LFSR type, 100 independent runs were performed, each with 250 training epochs. Fig. 4 shows the mean test accuracy and Table 3 shows the results for mean, maximum, minimum, 95%-percentile and 5%-percentile test accuracy. With a 16-bit LFSR employed during training of the CTM, we measured a mean test accuracy of 99.99% and a minimum test accuracy of 99.34%.

The performance of the CTM accelerator, as shown in Table 3, is on par with the results from the SW implementation in [3], as long as the lengths of the LFSRs are 16 or greater. Interestingly, all LFSR types obtain a MAX test accuracy of 100%. However, with shorter LFSRs one can note the significant degradation in performance, especially for the MIN and 5% percentile results.

We measured the power consumption of the FPGA by monitoring the voltage across an external resistor on the Zybo Z-20 board. The current through this resistor was a direct representation of the current draw of the FPGA. Although the accuracy of these measurements has a wide tolerance, relative measurements are considered to give a good indication of the differences in power consumption for various operating modes. In idle mode the total FPGA power consumption was measured to 2.522 W.

The increase in power consumption, from idle mode, to inference and training modes, was 7 mW and 187 mW respectively. If we consider the total power consumption of the FPGA, including the system processor and interface modules, the energy required is approximately 0.6 μJ per classification and 3.7 μJ per training example. Based on the power estimation tool in *Vivado*, about 90% of the power consumption is associated with the FPGA system processor.

In this work, the main target was to demonstrate a HW implementation of the CTM and to explore architectural trade-offs. As the accelerator operates on a custom-made dataset, direct comparison with other solutions is challenging. However, based on the achieved utilization of HW resources and the obtained energy consumption, we believe there is great potential for designing CTM accelerators with high performance operating on larger images. A significant reduction in power consumption can be expected by utilizing either (i) a low-power FPGA with a significantly simpler microcontroller than the one used in this work, (ii) a low-power FPGA in combination with a separate low-power microcontroller, or (iii) an ASIC solution.

We anticipate that scaling up the current design to support 28 × 28 images, for, e.g., the MNIST dataset, approximately 370 clock cycles will be required per classification. With an operating frequency of 40 MHz, this would correspond to about 1.1×10^5 classifications per second. Here we assume that a convolution window of 10 × 10 is utilized. In addition, booleanization of the greyscale MNIST images

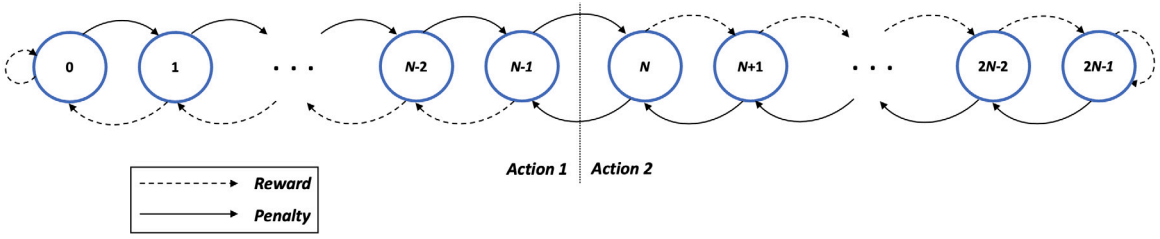


Fig. 5. A Tsetlin Automaton (TA) for two-action environments [9].

Table 3
Measured classification performance of the CTM accelerator, for varying LFSR bit lengths. For each LFSR type, 100 independent runs were performed, each with 250 training epochs.

LFSR length (bits)	6	7	8	9	10	12	14	16	18	24
Mean test accuracy (%)	84.51	96.05	99.67	99.03	99.62	99.42	99.48	99.99	99.95	99.88
Test accuracy MAX (%)	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Test accuracy MIN (%)	56.38	86.87	90.95	82.46	91.17	81.81	94.49	99.34	98.34	97.84
Test accuracy 95% perc. (%)	100.00	99.79	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Test accuracy 5% perc. (%)	64.99	89.80	99.38	94.96	98.00	95.03	94.49	99.88	99.80	98.90

must be performed before the samples are fed to the accelerator. Simple thresholding of the pixel values can be employed for MNIST classification by the CTM [3].

6. Conclusion

In this work, we have implemented a self-contained CTM accelerator for two-dimensional pattern classification on an FPGA platform, with full support for both training and inference. To the authors' knowledge, this is the first fully functional HW implementation of a CTM-based solution. The accelerator achieves a classification rate of 4.4 MFPS when operating at 40 MHz on 4 × 4 Boolean images. It achieves 99.9% mean test accuracy for the *noisy 2-dimensional dataset*, with 40% noise present in the training data labels, which matches the original SW solution. LFSR random number generators with a length of 16 bits or greater are required to obtain this performance.

Several architectural trade-offs are identified, and one can increase the classification and the training rates significantly by further parallelization. As our solution does not require complex arithmetic, in contrast to DNN solutions, we believe CTM-based solutions for image classification show great potential for low power applications. This work paves the way for future CTM-based HW solutions that can operate on more complex patterns and images. Our future research includes scaling the CTM accelerator to operate on the MNIST and CIFAR-10 datasets, and implementations in FPGAs and ASICs.

Data availability

Data will be made available on request

Appendix. Basic operation of the TM

The input to a TM [1] is a feature vector X consisting of o propositional Boolean variables, $x_u \in \{0,1\}$, $u = 0,\dots,o-1$. A new input vector L with in total $2o$ literals is formed by appending the negation of the variables to the input: $L = [x_0, \neg x_0, x_1, \neg x_1, \dots, x_{o-1}, \neg x_{o-1}]$.

A TM consists of several teams of Tsetlin Automata (TAs) that operates on the literals [1]. Each team of TAs forms a discriminative conjunctive clause by including or excluding literals. There are m clauses, c_j, where $j \in \{0,\dots,m-1\}$, and m is a user specified even integer. The odd indexed clauses are defined with positive polarity, while the even indexed ones have negative polarity.

The basic building block of the TM is the Tsetlin Automaton (TA), of two-action type. Its *include/exclude* decision for a given literal is achieved during the learning process.

Fig. 5 shows the structure of a single TA with $2N$ states. Action 2 (*include*) is employed if the TA is in one of the states from N to $2N-1$, while the states 0 to $N-1$ result in Action 1 (*exclude*). The TA is a finite-state-machine and in hardware it can typically be implemented as a binary up-/down-counter.

The output of a single clause, c_j is given by:

$$c_j = \bigwedge_{k \in I_j} l_k, \tag{9}$$

where l_k is the literal with index k, and k belongs to $I_j \subseteq \{0,\dots,2o-1\}$. I_j denotes the set of indexes of all the TAs that select action *include* in c_j. See Fig. 6(a).

In a basic two-class TM, classification is given by

$$\hat{y} = \begin{cases} 1 & \text{if } v \geq 0, \\ 0 & \text{if } v < 0. \end{cases} \tag{10}$$

where the output sum, v, is defined in Eq. (11). See Fig. 6(b).

$$v = \sum_{j=0}^{(m/2)-1} (c_{2j+1} - c_{2j}) \tag{11}$$

A q-class TM, as shown in Fig. 6(c), is constituted by several TMs, one for each class, from 0 to $q-1$. In this case, the final decision is made by an *argmax* operator that classifies the input data according to the highest vote sum [1].

A TM learns online, processing one training example (X, y) at a time. Learning takes place through a novel finite state learning automata game that coordinates the collective of TAs and leverages resource-allocation and frequent pattern mining principles [1]. Feedback mechanisms are employed and each TA is given either a *reward* or a *penalty*. If a TA does not receive *reward* nor *penalty*, the effective feedback is *inaction*, leaving the state unaffected. In Fig. 5, only the TA's state transitions related to *reward* and *penalty* are shown.

If a *reward* is applied, the TA will move deeper, i.e. towards state 0 or $2N-1$ depending on the action. With *penalty* the TA will move towards the center and will eventually switch to the other action.

During learning, there are two types of feedback: Type I and Type II. Table 4 shows the update probabilities and conditions for Type I feedback while Table 5 shows those for Type II feedback.

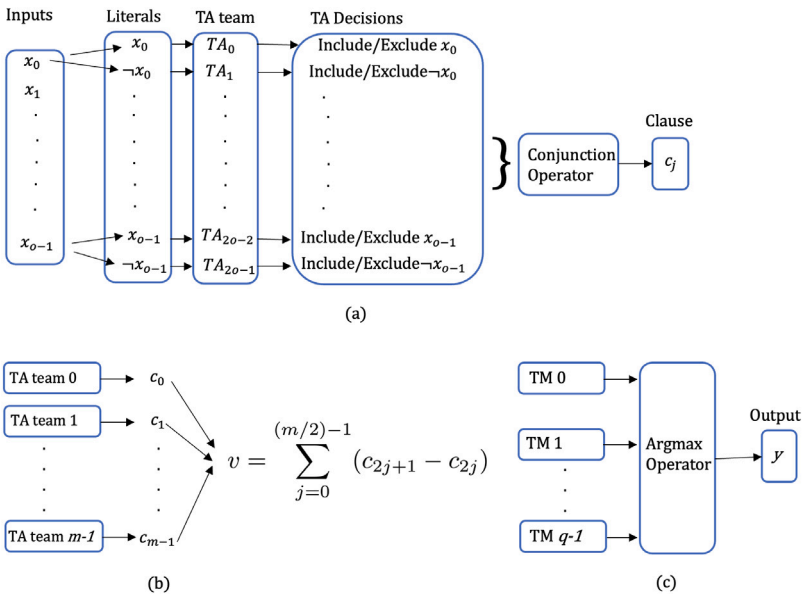


Fig. 6. (a) A TA team forms the clause c_j. (b) A two-class TM with m clauses. (c) A q-class TM [1].

Table 4
Type I feedback conditions and update probabilities. The feedback is for a single TA that decides whether to include or exclude a given literal l_k into c_j. NA means not applicable. s is a hyper-parameter greater than 1.

Clause value (c_j)		1		0	
Literal value (l_k)		1	0	1	0
TA: Include literal	P(Reward)	$\frac{s-1}{s}$	NA	0	0
	P(Inaction)	$\frac{1}{s}$	NA	$\frac{s-1}{s}$	$\frac{s-1}{s}$
	P(Penalty)	0	NA	$\frac{1}{s}$	$\frac{1}{s}$
TA: Exclude literal	P(Reward)	0	$\frac{1}{s}$	$\frac{1}{s}$	$\frac{1}{s}$
	P(Inaction)	$\frac{1}{s}$	$\frac{s-1}{s}$	$\frac{s-1}{s}$	$\frac{s-1}{s}$
	P(Penalty)	$\frac{s-1}{s}$	0	0	0

Table 5
Type II feedback conditions and update probabilities. The feedback is for a single TA that decides whether to include or exclude a given literal l_k into c_j. NA means not applicable.

Clause value (c_j)		1		0	
Literal value (l_k)		1	0	1	0
TA: Include literal	P(Reward)	0	NA	0	0
	P(Inaction)	1.0	NA	1.0	1.0
	P(Penalty)	0	NA	0	0
TA: Exclude literal	P(Reward)	0	0	0	0
	P(Inaction)	1.0	0	1.0	1.0
	P(Penalty)	0	1.0	0	0

Type I feedback is given stochastically to clauses with positive polarity when $y = 1$ and to clauses with negative polarity when $y = 0$. Each clause, in turn, updates each of its TAs based on: (1) its output $c_j(X)$; (2) the action of the TA controlling the literal – *Include* or *Exclude*; and (3) the value of the literal l_k assigned to the TA. Type I feedback is governed by two rules, as can be seen from Table 4:

- **Type Ia feedback:** *Include* is rewarded and *Exclude* is penalized with probability $\frac{s-1}{s}$ **if** $c_j(X) = 1$ **and** $l_k = 1$. This reinforcement is strong and makes the clause remember and refine the pattern it recognizes in X.
- **Type Ib feedback:** *Include* is penalized and *Exclude* is rewarded with probability $\frac{1}{s}$ **if** $c_j(X) = 0$ **or** $l_k = 0$. A large s will typically result in clauses with more literals.

Type II feedback is given stochastically to clauses with positive polarity when $y = 0$ and to clauses with negative polarity when $y = 1$. It penalizes *Exclude* actions with probability 1 **if** $c_j(X) = 1$ **and** $l_k = 0$. Thus, this feedback combats false positive outputs. Table 5 shows the update probabilities and conditions for Type II Feedback.

Resource allocation: To ensure that clauses distribute themselves across different frequent patterns, the learning procedure includes a mechanism for resource allocation. For any input X, the probability of updating a clause gradually drops to zero as the TM output sum in Eq. (11) approaches a user-configured target/hyperparameter T. A higher T increases the robustness of learning by allocating more clauses to learn each sub-pattern [24]. Optimum settings of m, T and s are dependent on the specific ML problem.

Eq. (12) shows the update probability, p_j^+, for clauses of the Target Class, while Eq. (13) gives the clause update probability for the Negative Target Class.

$$p_j^+ = \begin{cases} 1 & \text{with probability } \frac{T-\max(-T,\min(T,v))}{2T}, \\ 0 & \text{otherwise.} \end{cases} \tag{12}$$

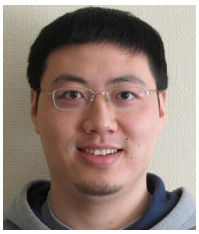
$$p_j^- = \begin{cases} 1 & \text{with probability } \frac{T+\max(-T,\min(T,v))}{2T}, \\ 0 & \text{otherwise.} \end{cases} \tag{13}$$

References

[1] O.-C. Granmo, The Tsetlin machine – a game theoretic bandit driven approach to optimal pattern recognition with propositional logic, 2018, arXiv e-prints.
[2] A. Wheeldon, R. Shafik, T. Rahman, J. Lei, A. Yakovlev, O.-C. Granmo, Learning Automata Based Energy-Efficient AI Hardware Design for IoT Applications: Learning Automata Based AI Hardware, Royal Society Publishing, 2020.
[3] O.-C. Granmo, S. Glimsdal, L. Jiao, M. Goodwin, C. Omlin, G. Berge, The convolutional Tsetlin machine, 2019, arXiv e-prints.
[4] S. Glimsdal, O.-C. Granmo, Coalesced multi-output Tsetlin machines with clause sharing, 2021, arXiv e-prints.
[5] K. Abeyrathna, B. Bhattarai, M. Goodwin, S. Gorji, O.-C. Granmo, L. Jiao, R. Saha, R. Yadav, Massively parallel and asynchronous Tsetlin machine architecture supporting almost constant-time scaling, in: Proceedings of the 38th International Conference on Machine Learning, Volume 139, in: Virtual, PMLR, 2021, pp. 10–20.
[6] R. Yadav, L. Jiao, O.-C. Granmo, M. Goodwin, Human-level interpretable learning for aspect-based sentiment analysis, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 35, 2021, pp. 14203–14212.
[7] B. Bhattarai, O.-C. Granmo, L. Jiao, Word-level human interpretable scoring mechanism for novel text detection using Tsetlin machines, Appl. Intell. (2022).

[8] R. Yadav, L. Jiao, O.-C. Granmo, M. Goodwin, Robust interpretable text classification against spurious correlations using AND-rules with negation, in: Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22), 2022, pp. 4439–4446, http://dx.doi.org/10.24963/ijcai.2022/616.
[9] X. Zhang, L. Jiao, O.-C. Granmo, M. Goodwin, On the convergence of Tsetlin machines for the IDENTITY- and NOT operators, IEEE Trans. Pattern Anal. Mach. Intell. (2021).
[10] L. Jiao, X. Zhang, O.-C. Granmo, K. Abeyrathna, On the convergence of Tsetlin machines for the XOR operator, IEEE Trans. Pattern Anal. Mach. Intell. (2022).
[11] L. Jiao, X. Zhang, O.-C. Granmo, On the convergence of Tsetlin machines for the AND and the OR operators, 2021, arXiv preprint.
[12] Y. Umuroglu, N. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, K. Vissers, FINN: A framework for fast, scalable binarized neural network inference, in: International Symposium on Field-Programmable Gate Arrays, 2016.
[13] F. Conti, P. Schiavone, L. Benini, XNOR neural engine: a hardware accelerator IP for 21.6 fJ/op binary neural network inference, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 37 (2018) 2940–2951.
[14] A. Mauro, F. Conti, P. Schiavone, D. Rossi, L. Benini, Always-on 674μW@4GOP/s error resilient binary neural networks with aggressive SRAM voltage scaling on a 22-nm IoT end-node, IEEE Trans. Circuits Syst. I. Regul. Pap. 67 (2020) 3905–3918.
[15] P. Knag, G. Chen, H. Sumbul, R. Kumar, S. Hsu, A. Agarwal, M. Kar, S. Kim, M. Anders, H. Kaul, R. Krishnamurthy, A 617-TOPS/W all-digital binary neural network accelerator in 10-nm finfet CMOS, IEEE J. Solid-State Circuits 56 (2021) 1082–1092.
[16] A. Kosuge, M. Hamada, T. Kuroda, A 16 nj/classification FPGA-based wired-logic DNN accelerator using fixed-weight non-linear neural net, IEEE J. Emerg. Sel. Top. Circuits Syst. 11 (2021) 751–761.
[17] D. Miyashita, S. Kousai, T. Suzuki, J. Deguchi, A neuromorphic chip optimized for deep learning and CMOS technology with time-domain analog and digital mixed-signal processing, IEEE J. Solid-State Circuits 52 (2017) 2679–2689.
[18] D. Bankman, L. Yang, B. Moons, M. Verhelst, B. Murmann, A. always on, 3.8 μJ/86% CIFAR-10 mixed-signal binary CNN processor with all memory on chip in 28-nm CMOS, IEEE J. Solid-State Circuits 54 (2019) 158–172.
[19] S. Venkataramanaiah, S. Yin, Y. Cao, J.-S. Seo, Deep neural network training accelerator designs in ASIC and FPGA, in: 2020 International SoC Design Conference (ISOCC), 2020, pp. 21–22, http://dx.doi.org/10.1109/ISOCC50952.2020.9333063.
[20] C. Banbury, V. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau, U. Thakker, A. Torrini, P. Warden, J. Cordaro, G. Guglielmo, J. Duarte, S. Gibellini, V. Parekh, H. Tran, N. Tran, N. Wenxu, X. Xuesong, Mlperf tiny benchmark, 2021, http://dx.doi.org/10.48550/arXiv.2106.07597, arXiv:2106.07597.
[21] A. Bakar, T. Rahman, A. Montanari, J. Lei, R. Shafik, F. Kawsar, Logic-based intelligence for batteryless sensors, in: Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications, HotMobile '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 22–28, http://dx.doi.org/10.1145/3508396.3512870.
[22] UCI machine learning repo, 1988, https://archive.ics.uci.edu/ml/datasets/iris.
[23] CTM github repository, 2019, https://github.com/cair/pyTsetlinMachine/tree/master/examples.
[24] K. Abeyrathna, O.-C. Granmo, M. Goodwin, Extending the tsetlin machine with integer-weighted clauses for increased interpretability, IEEE Access 9 (2021) 8233–8248.
[25] J. Vitter, Random sampling with a reservoir, ACM Trans. Math. Software 11 (1) (1985).
[26] S. Haykin, Communication Systems, fourth ed., John Wiley & Sons, Inc., New York, 2001.
[27] D. Sarwate, M. Pursley, Crosscorrelation properties of pseudorandom and related sequences, Proc. IEEE 68 (1980) 593–619.

Svein Anders Tunheim is a Ph.D. research fellow at the University of Agder at Centre for Artificial Intelligence Research (CAIR). He completed his M.Sc. degree in Electrical Engineering at The Norwegian University of Science and Technology (NTNU) in 1991. From 1992 to 1996 he worked as research scientist at SI (Senter for Industriforskning) and SINTEF within the field of mixed-signal integrated circuits (ICs). He was co-founder and Chief Technology Officer at Chipcon, a global supplier of low-power radio frequency ICs and radio protocols. From 2006 to 2008 he worked at Texas Instruments Norway as Technical Director for the Low Power Wireless product line. Currently, at CAIR, he is working on low-power hardware implementations of machine learning systems based on the Tsetlin Machine. He is a Senior Member of IEEE.



Prof. Lei Jiao received the B.E. degree in telecommunications engineering from Hunan University, Changsha, China, in 2005, the M.E. degree in communication and information system from Shandong University, Jinan, China, in 2008, and the Ph.D. degree in information and communication technology from the University of Agder (UiA), Norway, in 2012. He is currently working as a Professor with the Department of Information and Communication Technology, UiA. His research interests include reinforcement learning, Tsetlin machine, resource allocation and performance evaluation for communication and energy systems.

Dr. Rishad Shafik is a Reader in Electronic Systems within the School of Engineering, Newcastle University, UK. Dr Shafik received his Ph.D., and M.Sc. (with distinction) degrees from Southampton in 2010, and 2005; and B.Sc. (with distinction) from the IUT, Bangladesh in 2001. He is one of the editors of the Springer USA book "Energy-efficient Fault-tolerant Systems". He is also author/co-author of 170+ IEEE/ACM peer-reviewed articles, with 4 best paper nominations and 3 best paper/poster awards. He recently chaired multiple international conferences/symposiums, UK-CAS2020, ISTM2022; guest edited a special theme issue in Royal Society Philosophical Transactions A; he is currently chairing 2nd International Symposium on the Tsetlin Machine (ISTM), 2023. His research interests include hardware/software co-design for energy-efficiency and autonomy.

Prof. Alex Yakovlev received the Ph.D. degree from the St. Petersburg Electrical Engineering Institute, St. Petersburg, USSR, in 1982, and D.Sc. from Newcastle University, UK, in 2006. He is currently a Professor of Computer Systems Design, who founded and leads the Microsystems Research Group, and co-founded the Asynchronous Systems Laboratory, Newcastle University. He was awarded an EPSRC Dream Fellowship from 2011 to 2013. He has published more than 500 articles in various journals and conferences, in the area of concurrent and asynchronous systems, with several best paper awards and nominations. He has chaired organizational committees of major international conferences. He has been principal investigator on more than 30 research grants and supervised over 70 Ph.D. students. He is a fellow of the Royal Academy of Engineering, UK.

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 Ph.D. degree in 2004, both from the University of Oslo, Norway. In 2018 he created the Tsetlin machine, for which he was awarded the AI researcher of the decade by the Norwegian Artificial Intelligence Consortium (NORA) in 2022. Dr. Granmo has authored more than 160 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 Ph.D. students. Dr. Granmo is also a co-founder of NORA. Apart from his academic endeavours, he co-founded the companies Anzyz Technologies AS and Tsense Intelligent Healthcare AS.