Cyclostationary Random Number Sequences for the Tsetlin Machine

Svein Anders Tunheim[1], Rohan Kumar Yadav[1], Lei Jiao[1], Rishad Shafik[2], and Ole-Christoffer Granmo[1]

[1] Centre for Artificial Intelligence Research (CAIR),
University of Agder, Grimstad, Norway
{svein.a.tunheim, rohan.k.yadav, lei.jiao, ole.granmo}@uia.no
[2] Microsystems Group, School of Engineering, Newcastle University, UK
rishad.shafik@newcastle.ac.uk

Abstract. The Tsetlin Machine (TM) constitutes an emerging machine learning algorithm that has shown competitive performance on several benchmarks. The underlying concept of the TM is propositional logic determined by a group of finite state machines that learns patterns. Thus, TM-based systems naturally lend themselves to low-power operation when implemented in hardware for micro-edge Internet-of-Things applications. An important aspect of the learning phase of TMs is stochasticity. For low-power integrated circuit implementations the random number generation must be carried out efficiently. In this paper, we explore the application of pre-generated cyclostationary random number sequences for TMs. Through experiments on two machine learning problems, i.e., Binary Iris and Noisy XOR, we demonstrate that the accuracy is on par with standard TM. We show that through exploratory simulations the required length of the sequences that meets the conflicting tradeoffs can be suitably identified. Furthermore, the TMs achieve robust performance against reduced resolution of the random numbers. Finally, we show that maximum-length sequences implemented by linear feedback shift registers are suitable for generating the required random numbers.

Keywords: Machine learning · Tsetlin Machine · Cyclostationary random number sequences · Linear feedback shift registers

1 Introduction

The Tsetlin Machine (TM) is a novel machine learning (ML) algorithm that was introduced in 2018 [7]. TMs are based on propositional logic, leading to primarily Boolean operations. Such operations are in contrast to Deep Neural Networks (DNNs) where complex arithmetic is needed for Multiply-Accumulate Units (MACs). Furthermore, as propositional logic forms the basis of the algorithm, TMs have promising interpretability prospects. The unique features of TM make it suitable for low-energy hardware acceleration on Field Programmable

Gate Arrays (FPGA) and Integrated Circuits (ICs). Through compact implementation accelerated online training can be enabled for edge-nodes in Internet-of-Things (IoT) systems. To date TMs have been tested on tabular data, images, regression, natural language and speech, [3], and have shown competitive performance on several benchmarks in terms of accuracy, memory footprint and learning speed. For example, the convolutional TM (CTM) has obtained a peak test accuracy of 99.4% on MNIST, 96.31% on Kuzushiji-MNIST and 91.5% on Fashion-MNIST [8].

A key aspect of TMs is randomized choices during feedback based training. Effective randomization is crucial for avoiding deadlocks and overfitting using the training data. In the original software TM (Python/Cython) implementation [6], a randomization function is used to create a pseudo random number sequence [5]. The range of the numbers varies from 0 to RAND_MAX, where RAND_MAX is guaranteed to be at least 32767 (two bytes).

Python has a more advanced random sequence generator based on the Mersenne Twister [12]. This module produces 53-bit precision floating point numbers and has a period of $2^{19937} - 1$. It is also threadsafe and has been extensively tested.

Random sequence generation in hardware is, however, non-trivial. For low-complexity hardware implementation, the random sequence generation must be carried out efficiently. Storage of pre-generated random numbers in on-chip RAM/ROM is an alternative. However, this is only effective if relatively short sequences can be used. In a highly parallelized TM system, one will also need concurrent access to several independent random numbers, thus requiring many such sequences. Multi-word read capabilities from on-chip memory can reduce the number of sequences needed, but these require significant hardware resource allocation. Another alternative for random number generation is amplification of noise from an analog circuit module followed by analog-to-digital conversion. As true noise is the basis, it is possible to achieve very good stochastic properties [13]. However, mixed-signal design of the random number generator can lead to validation complexity as well as uncertainty due to the approximate nature of analog signals.

The concept of digital Pseudo Random Bit Sequences (PRBSs), typically implemented by Linear Feedback Shift Registers (LFSR) [9,14] is widely used for random number generation. LFSRs are suitable for hardware solutions as these can be implemented in digital and validation-friendly IC design flow. However, LFSRs usually require high level of switching depending on their sizes when the randomization process is active, which can consume non-negligible amount of energy.

In this paper, we aim to study an alternative randomization process suitable for hardware TM implementation. Core to this process is *whether a cyclostationary sequence of pre-generated random numbers could be applied as the source for random numbers in TMs.* Our main *contributions* are as follows:

– We evaluate if pre-generated cyclostationary sequences - with sequence address incremented for each lookup - can be applied for TM training and achieve accuracy similar to C/Python implementations.

– We study the impact of the number of elements in the sequence and therefore the tradeoffs between complexity and learning efficiency of the implementations.
– We evaluate the impact of the resolution (number of decimals) of the numbers in the sequence.

The remainder of this paper is organized as follows. Sec. 2 describes the TM architecture. Sec. 3 explores how cyclostationary sequences of random numbers can be applied to the TM. Sec. 4 details our experiment results together with discussions before we conclude in the final section.

2 Review of the Tsetlin Machine

A TM consists of several teams of Tsetlin Automata (TAs) that operates on literals, i.e., Boolean inputs and their complements. Each team of TAs forms a discriminative conjunctive clause by including or excluding literals as shown in Fig. 1(a). There are m clauses, c_j, where $j = 1, \ldots, m$, and m is an even integer. Half of the clauses, typically the odd numbered ones, are defined as positive, and the other half, the even numbered ones, are defined as negative, see Fig. 1(b). The outputs of the two groups of clauses are assembled in a majority voting unit to decide for the final classification, as shown in Fig. 1(c).

The TAs employed in a TM are of two-action type, i.e., a Tsetlin Automaton will either *include* or *exclude* a literal from a conjunctive clause. This is achieved during the learning process. Fig. 2 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 $N+1$ to $2N$, while the states 1 to N result in Action 1 (*exclude*).

The input to the TM is a feature vector $X = [x_1, x_2, \cdots, x_o]$ consisting of o propositional Boolean variables, $x_u \in \{0,1\}^o$, $u = 1, \ldots, o$. The negation of the variables are appended to the input forming a new input vector $\mathbf{L}$ with in total $2o$ literals: $[x_1, \neg x_1, x_2, \neg x_2, ..., x_o, \neg x_o]$. The output of a single clause, c_j is given by:

$$c_j = \bigwedge_{k \in I_j} l_k \tag{1}$$

Here l_k is the literal with index k, and k belongs to $I_j \subseteq \{1, ..., 2o\}$. I_j denotes the set of indexes of all the TAs that select action "include" in c_j.

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

$$\hat{y} = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases} \tag{2}$$

where the output sum, v, is defined in Eq. 3.

$$v = \sum_{j=1}^{m/2} c_{2j-1} - \sum_{j=1}^{m/2} c_{2j} \tag{3}$$

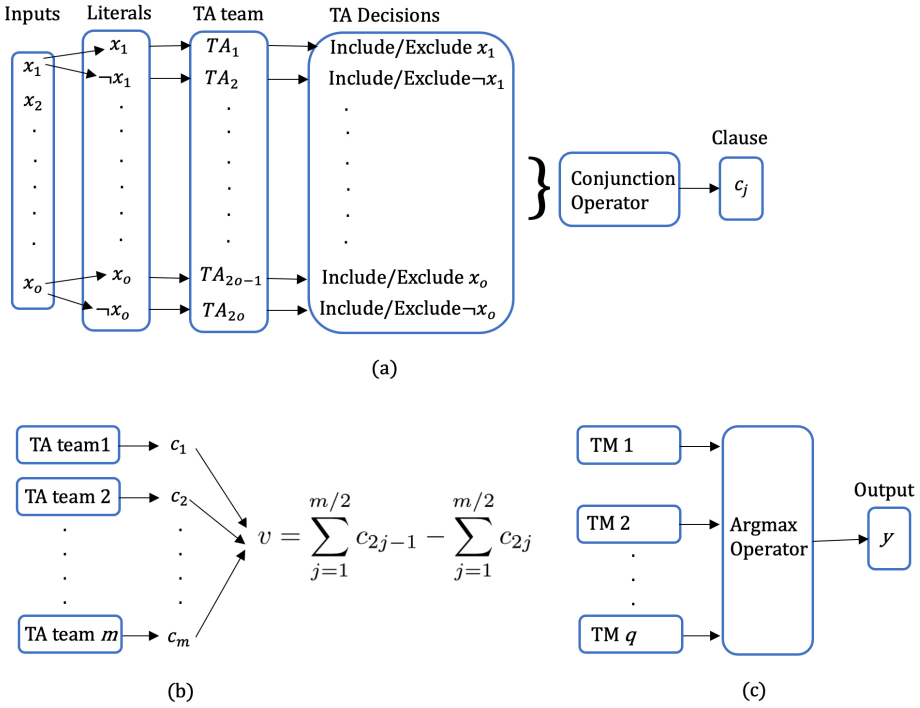

Fig. 1. (a) A TA team forms the clause c_j (b) A two-class TM with m clauses. (c) A q-class TM [7].

A multiclass TM is constituted by several TMs, one for each class, 1 to q. As shown in Fig. 1(c), the final decision is made by an argmax operator that classifies the input data according to the highest vote sum [7].

Learning takes place in the TM through a novel finite state learning automata game [7]. It coordinates the collective of TAs and leverages resource-allocation and frequent pattern mining principles. Feedback mechanisms are employed and gives each TA either a *reward* or *penalty*, depending on the training input, the individual clause outputs and the TM output sum, as shown in Fig. 2. If a reward is applied, the TA will move deeper, i.e. towards state 1 or $2N$ depending on the action. With penalty the TA will move towards the center and will eventually jump to the other side of the action. In addition to the number of clauses, m, the hyperparameters T (Threshold value) and s determine the stochastic learning characteristics. As explained in [7], T decides the clause update probability. A higher T increases the robustness of learning by allocating more clauses to learn each sub-pattern [1], while greater values of s stimulates a TA team to include more literals in the clause [7]. Optimum setting of m, T and s is dependent on the specific ML problem.

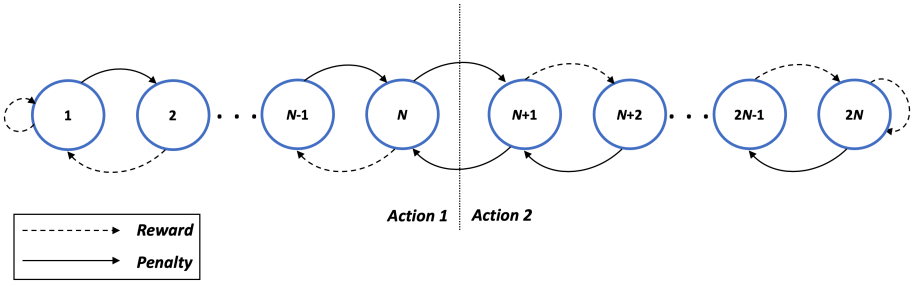

Fig. 2. A Tsetlin Automaton for two-action environments [18].

3 Proposed Random Number Generator in TM

In this section, we first give an overview of where stochasticity is employed in the TM algorithm. We then describe our proposed solution for applying the required random numbers to the TM. Thereafter, we elaborate how this solution can be implemented by LFSRs.

3.1 Randomization in Tsetlin Machine

Stochasticity is required by TM during training to avoid deadlocks and overfitting. Furthermore, the randomness helps the TM to allocate pattern recognition resources in an efficient way [7]. Stochasticity is applied during training at several different stages of the algorithm:

- Clause selection for update, based on the hyperparameter T, the class sum v, the clause type (positive/negative), and literal value l_k [7].
- TA update, based on the s hyperparameter [7].
- Patch selection for CTM [8]: A clause may output 1 for several patches during the convolution. TM randomly picks a patch among the ones that made the clause evaluate to 1 and trains the clause accordingly.
- For multiclass TM/CTM [1], [7]: In addition to training the *target class*, TM randomly selects a different class, the *negative target*, to train against, for a given training example.
- Epoch level: A TM version [15] features randomly (in a user-specified percentage) dropping of the clauses per training epoch. This produces more distinct and well-structured patterns that improve the performance and the learning robustness. During inference all literals are utilized. In addition, for each training epoch, the data should be randomly reshuffled [7], which increases robustness when operating on new input data.
- In the "arbitrarily deterministic TM" [2], the update of a TA only occurs every d'th time with a probability of 0.5. Here, d is an integer hyperparameter.

The random numbers generated for TMs should follow uniform probability distribution, according to [7], with range $[0, 1)$. TMs have been implemented in various programming languages such as Python, C and CUDA utilizing these languages' random number generators, and a huge amount of random numbers are generated for training. However, those approaches are not appropriate for low-power hardware platforms due to their complexity. In addition, the generation of random numbers can be energy and computational expensive. We therefore study below the possibility to simplify the generating process, and evaluate its impact on the performance of TM.

3.2 Employing pre-generated random number sequences for TM

To simplify the procedure of generating random numbers, we utilize a *pre-generated sequence* of random numbers in the range $[0, 1)$ with uniform probability distribution, instead of generating a random number every time. The sequence is *cyclostationary*. That is, it repeats itself when the last element has been read. This implies that the statistical properties of the sequence vary *cyclically* with time.

The `numpy.random.uniform` Python routine [10] was adopted to generate the numbers in the sequence. Sequences of various lengths were generated and different number of decimals of the random numbers were tested. For our experiments, we used a single cyclostationary sequence for all the random number accesses needed in the TM algorithm. The TM Python/Cython code was modified the following way:

- The *rand()* function was replaced by a lookup to a pre-generated sequence of random numbers, addressed by an index counter.
- For each sequence access the index counter was incremented.
- When reaching the last number in the sequence, the index counter was reset, thus re-accessing the sequence from the beginning.

3.3 Generating cyclostationary random number sequences with LFSRs

In addition to the pre-generated sequences, to accommodate an effective implementation of random number generators in hardware, we adopt and study the potential of *Linear Feedback Shift Registers* (LFSRs) [14] for TM. An LFSR consists of D-flip-flops connected in series, as a shift register, with feedback from select D flip-flop outputs, so-called "taps". The taps are XOR'ed and fed to the input of the shift register. Fig. 3 shows a 7-bit LFSR implementation where the taps 7 and 6 are XORed and fed to the input of the first D-flip-flop. We can describe this by the corresponding feedback polynomial $x^7 + x^6 + 1$.

By proper selection of taps, one can implement maximum length sequences (MLS) with the following properties [14]:

- The period is $2^N - 1$, where N is the length of the shift register.

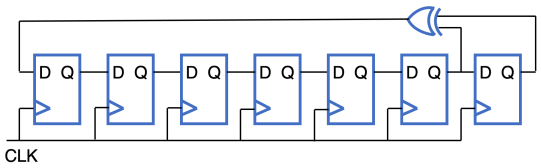

Fig. 3. A 7-bit LFSR.

- In each period of an MLS, the number of 1s is always one more than the number of 0s. This is called the *balance property*.
- A *run* is a group of consecutively following 1s or 0s. Among the runs of 1s and 0s in each period of an MLS, one half of the runs of each kind are of length one, one-fourth of length two, one-eight of length three, and so on as long as these fractions represent meaningful number of runs. This is called the *run property*.
- The *autocorrelation property* of an MLS is periodic and binary-valued [14], as shown in Eq. (4). Higher N implies lower autocorrelation, which in general is a desired property of random number sequences.

$$R(n) = \begin{cases} 1, & n = 0 \\ -\frac{1}{2^N - 1}, & 0 < n < 2^N - 1 \end{cases} \tag{4}$$

For the NoisyXOR problem, to be detailed in Subsection 4.1, we evaluated three different LFSRs based on the following MLS feedback polynomials [11]:

- $x^{18} + x^7 + 1$
- $x^{16} + x^{12} + x^3 + x^1 + 1$
- $x^{12} + x^6 + x^4 + x^1 + 1$

The above sequences have lengths of 262143, 65535 and 4095 respectively and were chosen based on the results from the general experiments, in Section 4. We used the open source LFSR Python software in [4] to generate the LFSR random numbers by reading out the internal register state per clock period. Furthermore, with this software we also verified the general MLS characteristics of these sequences, as described earlier in this section. The random numbers generated by the LFSRs were scaled to the range [0,1), and the resolution tested were floored to 2, 3 and 4 decimals. This corresponds to binary number representations of about 7, 10 and 14 bits respectively. For illustration purposes, Fig. 4 shows an excerpt from the sequence generated by the 18 bit LFSR. The numbers here are scaled to the range $[0, 1)$, and the sequence elements from 60000 to 60100 are shown.

4 Experimental Results and Discussion

To study the impact of different types of sequences on the performance of TM, experiments on two different ML problems were performed: The *NoisyXOR* and the *Binary Iris* datasets [7], [6].

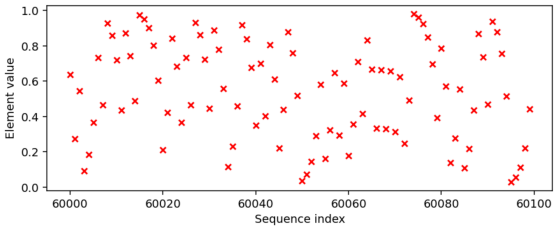

Fig. 4. Excerpt from sequence generated by the 18-bit LFSR.

4.1 The Noisy XOR Dataset with Non-informative Features

The artificial dataset of Noisy XOR with non-informative features was one of the original datasets [6] used for testing the vanilla TM [7]. The dataset consists of $10,000$ examples, and there are twelve Boolean inputs and one Boolean output. Ten of the inputs are completely random while two inputs follow the XOR-relation. Of the dataset, 50% is adopted for test and the other 50% for training. A high level of noise has been introduced in the training dataset by inverting 40% of the outputs. The motivation for this was to examine robustness of the TM [7]. The dataset is intended to uncover "blind zones" caused by XOR-like relations [7].

For comparison, the configuration of TM in this experiment is kept exactly the same as in [7]. The architecture is configured with 20 clauses and an s-value of 3.9, the threshold T value of 15. Each TA is allocated 100 states. We ran the TM algorithm for 200 epochs for each training experiment; the training data was reshuffled for each epoch. We performed 100 iterations of each experiment. Fig. 5 shows the mean test accuracy for different sequence lengths and different resolution of the pre-generated random numbers from these experiments.

In the original experiments reported in [7], the average of the test accuracy was 99.3%. The 5%-percentile, 95%-percentile, Min and Max accuracy were 95.9%, 100.0%, 91.6% and 100% respectively. Applying the pre-generated cyclostationary random number sequence for the NoisyXOR case, we observe that sequences with more than about 50k random numbers achieve mean test accuracy on par with the original results in [7], as shown in Figure 5. For example, for a sequence of 100k random numbers with 3 decimals, we obtained an average test accuracy of 99.5%, 5%-percentile of 97.7%, 95%-percentile of 100%, minimum accuracy of 93.1%, and maximum accuracy of 100%.

When it comes to resolution, applying a number with only 1 decimal significantly degrades the accuracy, independently of the sequence length. However, there are only minor accuracy differences between 2 and more decimals for sequence lengths greater than 50k elements. For sequence lengths of 100k and 200k, we also tested rounding down to the closest decimal of the chosen resolution, i.e. the numbers were "floored", as this reflects a close-to-hardware representation. In this case, comparing to the normal rounding showed negligible difference.

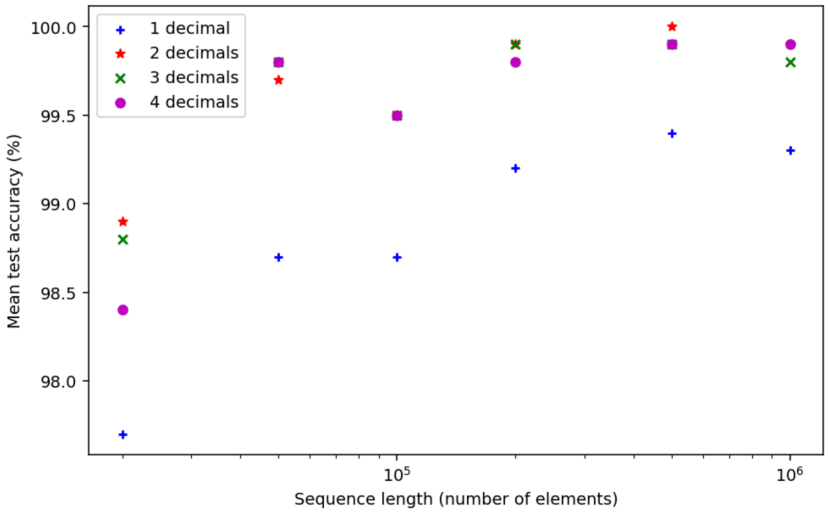

Fig. 5. Average accuracy for NoisyXOR test data versus sequence length and for different random number resolutions.

As the sequence needs to be long to achieve a comparable accuracy, i.e., over 50k numbers, storing it in on-chip RAM/ROM is not attractive. A sequence of 50k numbers with, e.g., 10 bits representation, would need 0.5Mbit storage. Furthermore, with highly parallel operation of a TM-based system it is desirable to operate several such sequences in parallel, requiring more hardware resources. For this reason, LFSRs are considered as better candidates for random number generators in TM-based hardware systems. It should be noted, however, that increased register lengths will result in higher power consumption due to the digital switching activity. Therefore, TM training circuitry will only be enabled during training and will be switched off to save power during inference mode. In an IC, clock and power gating can be employed to enable the LFSRs. Clause and TA updating can be implemented with a high degree of parallelism, and several LFSRs can operate concurrently. In this case, the different LFSRs should be seeded differently, i.e., their start conditions should be different.

In Table 1, it is shown that LFSR registers, with lengths from 16 and upwards, provide random numbers that do not degrade the mean test accuracy. The 12 bit LFSR corresponds to a sequence length of only 4095, and the accuracy degradation for this is notable and as expected.

For the results in Table 1, it should be noted that the sequences also included the startup transition part. More specifically, the sequences' probability distribution may not be precisely uniform. To test this further, we applied only 1/4 of the sequence for the 18 bit LFSR as the source for random numbers. In this case, we found that the mean accuracy was also high (99.9%). The reason for this is most likely the MLS's balance and run properties as shown in Section 3. Moreover, the quarter sequence also contains more than 50k elements.

Table 1. Mean accuracy for NoisyXOR test data versus sequence length for different number resolutions. MLS/LFSR-based number sequences. The random numbers are floored.

	2 decimals	3 decimals	4 decimals
18 bit LFSR	99.9	99.9	99.9
16 bit LFSR	99.9	99.9	99.8
12 bit LFSR	88.7	85.8	82.0

4.2 The Binary Iris Dataset

The Iris dataset is classical [16] and consists of only 150 examples. It was one of the datasets used when evaluating the vanilla TM performance [7]. Each example has four inputs and three possible outputs (classes). In [7], the dataset was converted into Boolean features the following way: Each input value was represented by a 4 bit number, i.e. the input sequence was 16 bits in total. This new dataset was denoted The Binary Iris Dataset. For training we used 80% of the dataset, and we randomly generated 100 different training and test partitions (ensembles). We adopted 300 clauses per class. The s-value was 3.0, the threshold T was 10 and each TA had 100 states. We ran the TM for 500 epochs for each data partition.

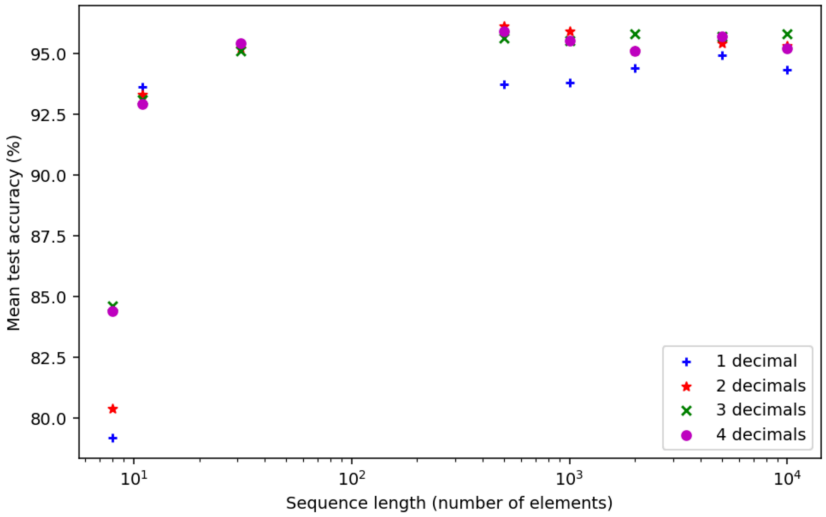

Fig. 6. Mean accuracy for Binary Iris test data for different sequence lengths and number resolutions.

In Fig. 6, it is shown the mean accuracy for Binary Iris test data for different sequence lengths and number resolutions. The original mean test data accuracy

was 95.0% [7]. As we can see from Fig. 6, even very short sequences provide good accuracy. For 11 elements and below we observed significant accuracy degradation. Similar to the NoisyXOR dataset, the resolution does not have a huge impact. However, with only one decimal there is a notable degradation. Thus, the chosen settings for the Binary Iris dataset are very robust with respect to the random number sequence.

The IC reported in [17] implements a TM-based classifier for the Binary Iris case. It the first reported chip based on the TM, and achieves 62.7Top/J during inference and 34.6Top/J during training. The chip employs one 8-bit LFSR per TA, and each LFSR is seeded differently. The accuracy of the IC was somewhat degraded compared with the original software version. The IC shows approximately 92.5% for the test accuracy. Our simulations show that even with very short sequence lengths, the TM operates robustly for this dataset. So the 8-bit LFSR with a sequence length of 255 should be sufficient. However, other IC implementation choices affect the test accuracy. Most important is the number of clauses applied per class, which for the IC in [17] was significantly less than for the results reported in [7].

The number of times a sequence is applied during one epoch scales approximately inversely proportional with the sequence length as expected. The number of sequence accesses depends on the amount of training examples and the general TM configuration (number of clauses, number of classes, T, and s). For the NoisyXOR case, a 50k element sequence is applied approximately 13.9 times during one epoch, while the Binary Iris only requires 0.9 times.

Comparing the results from these two data sets, we can conclude that cyclostationary sequences can be used by TM for training purpose. For NoisyXOR, the sequence should contain more than 50k elements, and the resolution of the random numbers should be minimum two decimals. For Binary Iris, the required length is significantly reduced. This indicates that the required sequence length is to a large degree dependent on the ML problem and the nature of the dataset. This shows that for any hardware implementations, if the application domain and the ML problem is known, the random sequence generator can be designed according to the nature of the problem. The approach may be tested in software implementations first, and then deployed accordingly in the hardware, which can best balance the complexity, performance, and power consumption. More specifically, one can modify the TM code by e.g., exchanging the random function calls with lookup to a pre-generated number sequence, and perform direct simulations of the effect of the sequence length and resolution.

5 Conclusions

Based on the empirical results, we conclude that pre-generated cyclostationary sequences of randomly generated numbers can be employed for TM training with test accuracy on par with vanilla TM. The required lengths of the sequences depend heavily on the ML problem. Using sequences with just sufficient lengths can reduce the complexity of the implementation of TM significantly. For hard-

ware implementations, applying cyclostationary sequences from LFSRs is an attractive and robust solution for the random number sequence generation.

References

1. Abeyrathna, K.D., Granmo, O.C., Goodwin, M.: Extending the Tsetlin machine with integer-weighted clauses for increased interpretability. IEEE Access **9**, 8233–8248 (2021), https://ieeexplore.ieee.org/document/9316190
2. Abeyrathna, K.D., Granmo, O.C., Shafik, R., Yakovlev, A., Wheeldon, A., Lei, J., Goodwin, M.: A multi-step finite-state automaton for arbitrarily deterministic tsetlin machine learning. Expert Systems, John Wiley & Sons Ltd. (2021), https://doi.org/10.1111/exsy.12836
3. Abeyrathna, K.D., Bhattarai, B., Goodwin, M., Gorji, S.R., Granmo, O.C., Jiao, L., Saha, R., Yadav, R.K.: Massively parallel and asynchronous tsetlin machine architecture supporting almost constant-time scaling. In: Meila, M., Zhang, T. (eds.) Proceedings of the 38th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 139, pp. 10–20. PMLR (18–24 Jul 2021), https://proceedings.mlr.press/v139/abeyrathna21a.html
4. Bajaj, N.: Nikeshbajaj/linear_feedback_shift_register: 1.0.6 (Apr 2021), https://doi.org/10.5281/zenodo.4726667
5. C library function - rand(): https://www.tutorialspoint.com/c_standard_library/c_function_rand.htm
6. Granmo, O.C.: Github TM repo. https://github.com/cair/TsetlinMachine
7. Granmo, O.C.: The Tsetlin machine – a game theoretic bandit driven approach to optimal pattern recognition with propositional logic. arXiv e-prints (2018), https://arxiv.org/abs/1804.01508
8. Granmo, O.C., Glimsdal, S., Jiao, L., Goodwin, M., Omlin, C.W., Berge, G.T.: The convolutional Tsetlin machine. arXiv e-prints (2019), https://arxiv.org/abs/1905.09688
9. Haykin, S.: Communication Systems, 4th. Edition. John Wiley & Sons, Inc. (2001)
10. Numpy library: https://www.numpy.org
11. Partow, A.: http://www.partow.net/programming/polynomials/index.html
12. Python 3.10.0 doc.: https://docs.python.org/3/library/random.html
13. Rahman, T., Shafik, R., Granmo, O.C., Yakovlev, A.: Resilient biomedical systems design under noise using logic based machine learning. Frontiers in Control Engg.: Adaptive, Robust and Fault Tolerant Control (in press, 2022), https://www.frontiersin.org/articles/10.3389/fcteg.2021.778118/abstract
14. Sarwate, D., Pursley, M.: Crosscorrelation properties of pseudorandom and related sequences. Proceedings of the IEEE **68**(5), 593–619 (1980)
15. Sharma, J., Yadav, R., Granmo, O.C., Jiao, L.: Human interpretable AI: Enhancing Tsetlin machine stochasticity with drop clause. arXiv (2021), https://arxiv.org/abs/2105.14506
16. UCI Machine Learning Repo.: https://archive.ics.uci.edu/ml/datasets/iris
17. Wheeldon, A., Shafik, R., Rahman, T., Lei, J., Yakovlev, A., Granmo, O.C.: Learning automata based energy-efficient AI hardware design for IoT applications: Learning automata based AI hardware. Royal Society Publishing (2020), https://royalsocietypublishing.org/doi/10.1098/rsta.2019.0593
18. Zhang, X., Jiao, L., Granmo, O.C., Goodwin, M.: On the convergence of tsetlin machines for the identity-and not operators. IEEE Transactions on Pattern Analysis and Machine Intelligence (2021). https://doi.org/10.1109/TPAMI.2021.3085591