An Alternate Feedback Mechanism for Tsetlin Machines on Parallel Architectures

Jordan Morris, Ashur Rafiev, Fei Xia, Rishad Shafik, Alex Yakovlev
μSystems design group
Newcastle University
Newcastle, UK
{jordan.morris, ashur.rafiev, fei.xia, rishad.shafik, alex.yakovlev}@newcastle.ac.uk

Andrew Brown
Dept. of ECS
University of Southampton
Southampton, UK
adb@ecs.soton.ac.uk

***Abstract*—This work proposes an alternative feedback mechanism for the Tsetlin Machine, a nascent machine learning algorithm that accepts binarized input data and uses propositional logic to identify and accumulate sub-patterns from a given entropy. The proposed method monitors and limits the included literals that contribute to the sub-patterns. This permits the algorithm to converge without requiring the class sum, the primary hurdle of a fully parallelized implementation. Empirical results from a custom RISC-V NoC cluster demonstrate up to a 36X reduction in wall-clock runtime for a 2.5% reduction in accuracy using the MNIST dataset. The proposed method outperforms the original feedback mechanism by 2% when the number of accumulated sub-patterns (clauses) are tightly constrained for the same dataset. This is achieved with a 1.8X reduction in wall-clock runtime.**

***Index Terms*—Tsetlin Machine, Parallel Architectures, RISC-V**

I. Introduction

The Tsetlin Machine [1] is a relatively nascent machine learning algorithm that uses binarized input vectors and derives sub-patterns from a given entropy. Binary literals from the input vector are accumulated in conjunctive clauses which in turn vote for a given outcome. These votes are then accumulated per class in order to resolve classification problems. This mechanism has been extended to other machine learning tasks such as regression [2] and text categorization [3], as well as being efficiently implemented in silicon [4].

In the original Tsetlin Machine algorithm, the accumulated class-wide clause votes (the class sums) are used within the feedback mechanism to stimulate sub-pattern optimisation and allow the algorithm to eventually converge. The drawback to this is that it requires inter-clause communication in an otherwise outright parallelizable algorithm. One proposed solution to this is to duplicate the class sums and attach these to the datapoints in an example pool [5]. This creates a freshness issue because as soon the include/exclude profile within a clause changes, the duplicated class sums must see the clause to know if their own copy requires updating. The method functions under the principle of *eventual consistency* as eventually, all clauses see all datapoints throughout several epoch iterations.

This work was supported under EPSRC Grant EP/N031768/1 (POETS). The corresponding author is Jordan Morris.

This work proposes an alternative algorithm that monitors and limits the literals that form the sub-pattern within a clause. This ”literal count” is then used to stimulate sub-pattern optimisation and converge the algorithm, replacing the class sum and allowing the clauses to run independently.

Section II provides an overview of the RISC-V FPGA platform used to test the algorithm. Section III describes behavioral vs. literal feedback and delineates the changes made to original Tsetlin Machine algorithm. Section IV describes the operation and thread layout of the parallel experiments. Section V provides results, discusses their implications and suggests future work. Section VI concludes the work presented.

II. The POETS Architecture

POETS [6] (Partially Ordered Event Triggered Systems) is both a development-model and concrete hardware-software stack for developing parallel applications. An overview of the POETS architecture may be seen in Figure 1.

The basic hardware unit consists of four cores sharing a mailbox, cache and floating point unit (FPU). This is known as a tile. Each core is instantiated as a customized 32-bit multi-threaded processor implementing a subset of the RV32IMF profile of the RISC-V instruction set, complete with 16 hardware threads. Each hardware thread is capable of scheduling multiple software threads.

Each FPGA board consists of 16 networked tiles arranged in a 4 X 4 matrix sharing 4GB of off-chip RAM. Four 10Gbps links are available for inter-board routing. Boards are arranged in a 3 X 2 matrix within a single box. External access for configuration and data entry/retrieval is provided via an x86 machine. The cluster currently consists of 8 boxes arranged in a 2 X 4 configuration. Ethernet links provide inter-box communication. The cluster therefore consists of 48 FPGAs providing 49152 networked hardware threads.

Lowest in the software stack is a custom overlay called Tinsel [7], which handles low level events and communication. Tinsel handles message acceleration via distributed hardware multicast [8] and core resource sharing. Tinsel also implements termination detection [9] driven by threads indicating when they have no more messages to send.

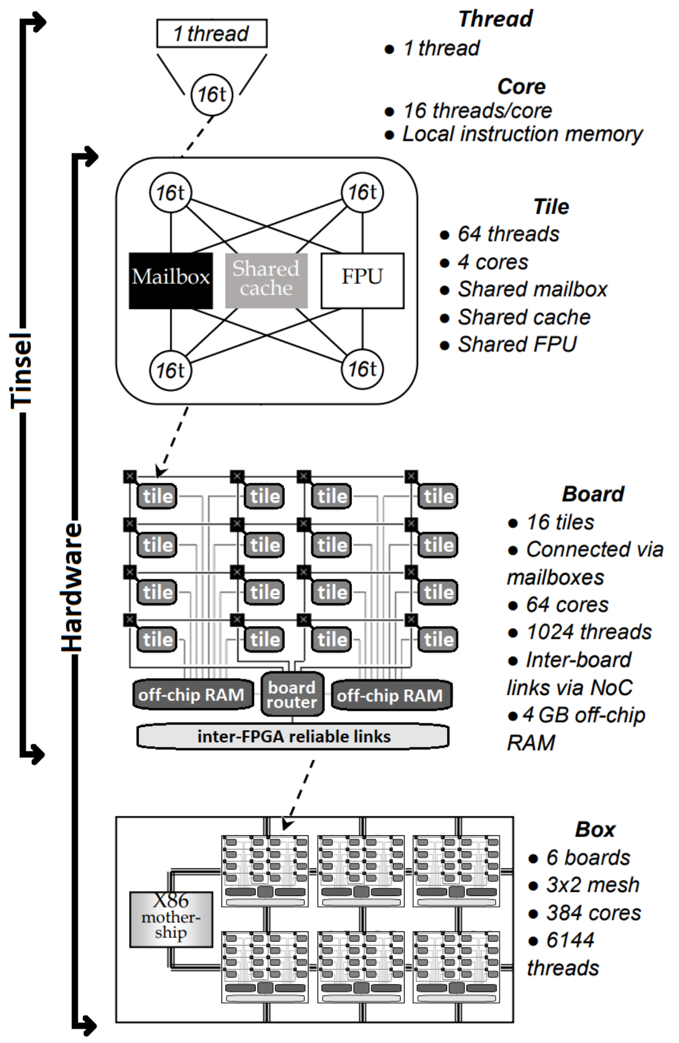

Fig. 1. POETS Architecture

III. Literal Limiting

A. Behavioral vs. Literal Feedback

The reward/penalty decisions in the original feedback specification may be transformed into simple promote/demote decisions based on the update operation to the tsetlin automata. This translates into simpler code.

B. Literal Limiting

The original Tsetlin Machine algorithm must be specified with a class sum threshold, T, as a hyper-parameter. This is used to limit the class sum values per datapoint, effectively limiting the maximum number of clauses that vote for or against a particular class for the provided datapoint, X_i. As the class sum approaches +T/-T, the overall feedback to the automata is reduced, allowing the algorithm to converge. Providing this limit also forces the sub-patterns to optimise for the entire dataset.

The alternative mechanism investigated in this work replaces T with a clause-local literal threshold, L. This effectively limits the sub-pattern size within each clause. This is then used to converge the algorithm and the feedback mechanisms are modified to force sub-pattern optimisation.

For this to function, the Tsetlin Automata must be initially excluded (to give an included literal count of zero) and therefore all automata are initialized to exclude on the include/exclude boundary. This is contrary to the original algorithm that randomly assigns include/exclude status on a non-inverted/inverted pair-wise basis.

Algorithm 1 provides high level overview of the proposed clause update process. After receiving a datapoint, the algorithm performs the standard conjunction calculation. The included literals for the datapoint are then counted. Literals that resolve to a '1' increment the literal count, those that provide a '0' decrement it. This is shown in Algorithm 2. The algorithm then chooses which feedback to perform using the same method as the original Tsetlin Machine algorithm (as shown in Algorithm 1).

If type I feedback is required, this is provided as shown in Algorithm 3. A random starting point amongst the automata within the clause is chosen. This is to prevent the mechanism favouring sub-patterns towards the beginning of the datapoint. The literal count is then clipped to the literal limit, L. The probability of receiving feedback is then calculated on an automata basis using the current clipped literal count, l_i^c, and literal limit, L. If feedback is provided, the clipped literal count is updated accordingly. If l_i^c equals L whilst the automata are updated, no further feedback will be provided and the feedback mechanism may be exited.

If type II feedback is required, this is provided as shown in Algorithm 4. This first checks if the clause resolves to '1'. If so, a random starting point amongst the automata within the clause is chosen as in the type I feedback mechanism. It then promotes excluded automata with the literal value '0' and checks whether the automata have crossed the boundary into include. On the first crossover, the mechanism updates the literal count and stops providing feedback. This prevents the literal count and sub-pattern from saturating with zeroes but still forces the conjunction for the current datapoint to resolve to zero, stimulating the sub-pattern to alter via the type I feedback mechanism.

The overall result is that the Tsetlin Machine converges when each clause has a sub-pattern of size L. If a clause resolves to '1' for an incorrect class, the type II feedback mechanism randomly adds a '0' onto the sub-pattern. The type I feedback mechanism either removes the '0' and retains the original sub-pattern or removes one or more '1's then removes the '0'. The '1's are then randomly re-added in different locations, updating the sub-pattern that the clauses represents.

IV. Thread Layout

Each clause in the Tsetlin Machine was provided its own thread on the cluster. Both training and inference datasets were pushed to separate storage threads on the FPGA fabric. This allowed for node-side multicast acceleration on the POETS architecture.

Algorithm 1 Decentralized Clause Update

```
Input: Example Pool P, clause $C_j$, polarity indicator $p_j \in$ {0,1}, sensitivity factor $s \in \{1,\infty\}$, literal count $l$, literal limit $L$
    Procedure: UpdateClause: P, $C_j$, $p_j$, $s$, $l$, $L$
1: $(X_i, y_i) \leftarrow$ ReceiveTrainingExample(P)
2: $C_j \leftarrow$ CalculateConjuction($X_i$)
3: $l_i \leftarrow$ SumIncludedLiterals($X_i$, $C_j$)
4: if $y_i$ xor $p_j$ then
5:     $C_j \leftarrow$ ModifiedTypeIIFeedback($X_i$, $C_j$, $l_i$)
6: else
7:     $C_j \leftarrow$ ModifiedTypeIFeedback($X_i$, $C_j$, $s$, $L$, $l_i$)
8: end if
```

Algorithm 2 Sum Included Literals

```
Input: literal $k$, literals per datapoint $K$, literal count $l$
    Procedure: SumIncludedLiterals: $k$, $K$, $l$
1: $l \leftarrow 0$
2: for $i = 0$ to $K$ do
3:     if literalIncluded($k_i$) then
4:         if $k_i$ = 1 then
5:             $l$++
6:         else
7:             $l$- -
8:         end if
9:     end if
10: end for
```

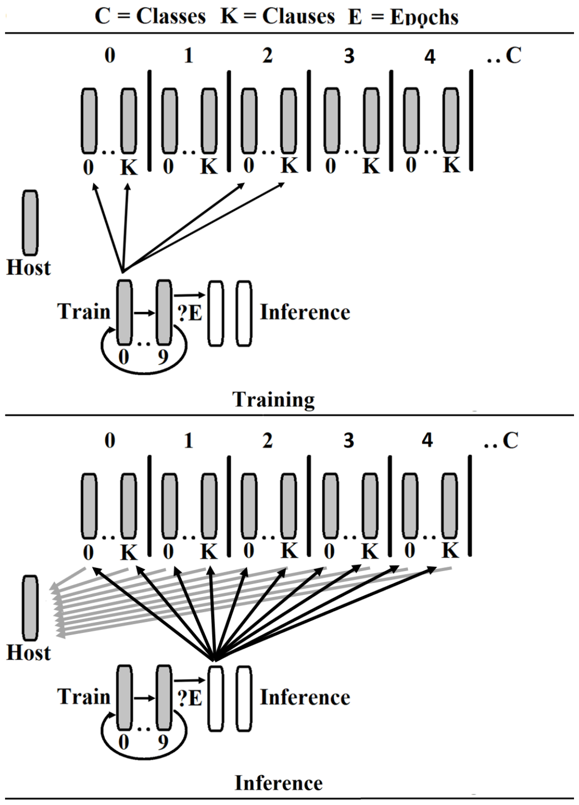

Fig. 2. Thread Layout

Algorithm 3 Modified Type I Feedback

```
Input: clause $C_j$, sensitivity factor $s \in \{1,\infty\}$, literal $k$, literal count $l$, literals per datapoint $K$, literal limit $L$, randomized literal location $irand$
    Procedure: ModifiedTypeIFeedback: $C_j$, $s$, $k$, $l$, $K$, $L$, $irand$
1: $irand \leftarrow$ rand() % $K$
2: for $i = 0$ to $K$ do
3:     $l_i^c \leftarrow$ clip($l_i$, $L$)
4:     if rand() < ($L$ - $l_i^c$) / $L$ then
5:         if $C_j$ and LiteralIsOne($k_{irand}$) then
6:             if rand() < (1 - (1 / $s$)) then
7:                 PromoteTA($k_{irand}$)
8:                 if NewlyIncluded($k_{irand}$) then
9:                     $l_i^c$++
10:                end if
11:            end if
12:        else
13:            if rand() < (1 / $s$) then
14:                DemoteTA($k_{irand}$)
15:                if NewlyExcluded($k_{irand}$) then
16:                    if LiteralIsOne($k_{irand}$) then
17:                        $l_i^c$- -
18:                    else
19:                        $l_i^c$++
20:                    end if
21:                end if
22:            end if
23:        end if
24:    end if
25:    $irand \leftarrow$ ($irand$ + 1) % $K$
26: end for
```

Algorithm 4 Modified Type II Feedback

```
Input: clause $C_j$, literal $k$, literals per datapoint $K$, randomized literal location $irand$
    Procedure: ModifiedTypeIIFeedback: $C_j$, $k$, $K$, $irand$
1: if $C_j$ = 1 then
2:     $irand \leftarrow$ rand() % $K$
3:     for $i = 0$ to $K$ do
4:         if LiteralIsZero($k_{irand}$) and LiteralExcluded($k_{irand}$)
           then
5:             PromoteTA($k_{irand}$)
6:             if LiteralIncluded($k_{irand}$) then
7:                 $l$- -
8:                 break
9:             end if
10:        end if
11:        $irand \leftarrow$ ($irand$ + 1) % $K$
12:    end for
13: end if
```

An overview of the thread layout may be seen in Figure 2. Training data was stored on 10 threads that mutlicast each datapoint to its own class and one other random class. Once all 10 threads had broadcast, a check was performed to determine if the required number of epochs was reached. If not, the training data was broadcast again. If so, two threads holding the inference data broadcast the datapoints to all classes. On performing the inference, each clause sent a simple message containing the clause output, class number, datapoint number and clause polarity to the host node. This then accumulated the votes and determined the highest class vote per datapoint. This entire process was event-driven and asynchronous. Messages were placed onto the network until network capacity was reached. This forced the application to become CPU bound, allowing all threads to run at maximum capacity in parallel during the entirety of operation.

V. Experiments

All runs were performed using the full MNIST datatset. The original algorithm was run on an Intel i9-7940X CPU @ 3.10GHz. The class sum threshold, T, was optimised in accordance with the number of clauses. The proposed algorithm was run on the POETS cluster with the RISC-V cores clocked at 210MHz. The literal limit hyper-parameter, L, was set at 50 for all runs. Both implementations were run without boosting the true positive feedback. Figure 3 shows the accuracy and wall-clock runtime over varying clause counts.
The results demonstrate a minor loss of 2.5% in accuracy for 40K clauses but a reduction in wall-clock runtime of 36X. The difference in accuracy is likely due to the superior ability of the original algorithm to identify outliers in the dataset. Conversely, the results show a 2% improvement in accuracy at 2K clauses with a runtime improvement of 1.8X. This is likely due to superior ability of the proposed algorithm to use all clauses in the classification process.
There are several avenues of further study from the experiments performed. Given that the proposed algorithm relies more on the type I feedback mechanism of including literals, boosting the true positive feedback is likely to have a significant impact on the algorithm. Type II feedback is always given in the proposed algorithm. Introducing a probabilistic guard to this may improve accuracy and runtime by eliminating the perpetual churn of sub-patterns. Alternatively, locking type II feedback for clauses with sub-patterns that appear in many datapoints would essentially 'lock-in' the most effective sub-patterns and force the remaining clauses to focus on outliers.

VI. Conclusion

This paper presented an alternative feedback mechanism for Tsetlin Machines, based on the monitoring and limiting of the literals that generate the sub-patterns. The experiments presented demonstrated that a 36X reduction in wall-clock runtime was possible for a 2.5% loss in accuracy in comparison to the original algorithm. Several areas of future research were suggested which have the potential to reduce this loss in accuracy and improve on the current wall-clock runtime reductions.

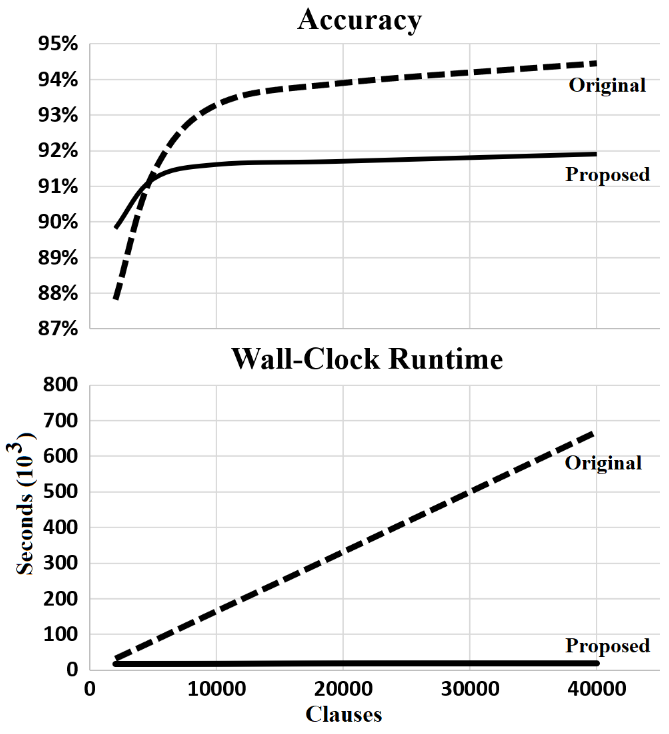

Fig. 3. Accuracy and Wall-Clock Time Over Clause Scaling

References

[1] O. C. Granmo. The tsetlin machine – a game theoretic bandit driven approach to optimal pattern recognition with propositional logic. *arXiv preprint arXiv:1804.01508*, 2018.

[2] K. D. Abeyrathna, O.-C. Granmo, X. Zhang, L. Jiao, and M. Goodwin. The regression tsetlin machine - a novel approach to interpretable non-linear regression. *Philosophical Transactions of the Royal Society A, 378*, 2020.

[3] G. T. Berge, O.-C. Granmo, T. Tveit, M. Goodwin, L. Jiao, and B Matheussen. Using the tsetlin machine to learn human-interpretable rules for high-accuracy text categorization with medical applications. *IEEE Access, 7:115134–115146*, 2019.

[4] A. Wheeldon, R. Shafik, T. Rahman, J. Lei, and A. Yakovlev. The tsetlin machine – a game theoretic bandit driven approach to optimal pattern recognition with propositional logic. *Philosophical Transactions of the Royal Society A*, 2020.

[5] K. D. Abeyrathna, B. Bhattarai, M. Goodwin, S. Gorji, O. C. Granmo, L Jiao, R. Saha, and Yadav. R. K. Massively parallel and asynchronous tsetlin machine architecture supporting almost constant-time scaling. *arXiv:2009.04861*, 2021.

[6] A. D. Brown, M. L. Vousden, A. D. Rast, G. M. Bragg, D. B. Thomas, J. R. Beaumont, M. F. Naylor, and A Mokhov. Poets: Distributed event-based computing-scaling behaviour. *Proc ParCo*, 2019.

[7] M Naylor, S. W. Moore, and D Thomas. Tinsel: a manythread overlay for fpga clusters. *IEEE International Conference on Field Programmable Logic and Applications*, 2019.

[8] M Naylor, S. W. Moore, D Thomas, J. R. Beaumont, S Fleming, and M Vousden. General hardware multicasting for fine-grained message-passing architectures. *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2021.

[9] M Naylor, S. W. Moore, A Mokhov, D Thomas, J. R. Beaumont, S Fleming, A. T. Markettos, T Bytheway, and A Brown. Termination detection for fine-grained message-passing architectures. *IEEE International Conference on Application-specific Systems, Architectures, and Processors*, 2020.