# Massively Parallel and Asynchronous Tsetlin Machine Architecture Supporting Almost Constant-Time Scaling

Kuruge Darshana Abeyrathna [*1] Bimal Bhattarai [*1] Morten Goodwin [*1] Saeed Rahimi Gorji [*1] Ole-Christoffer Granmo [*1] Lei Jiao [*1] Rupsa Saha [*1] Rohan Yadav [*1]

## Abstract

Using logical clauses to represent patterns, Tsetlin machines (TMs) have recently obtained competitive performance in terms of accuracy, memory footprint, energy, and learning speed on several benchmarks. Each TM clause votes for or against a particular class, with classification resolved using a majority vote. While the evaluation of clauses is fast, being based on binary operators, the voting makes it necessary to synchronize the clause evaluation, impeding parallelization. In this paper, we propose a novel scheme for desynchronizing the evaluation of clauses, eliminating the voting bottleneck. In brief, every clause runs in its own thread for massive native parallelism. For each training example, we keep track of the class votes obtained from the clauses in local voting tallies. The local voting tallies allow us to detach the processing of each clause from the rest of the clauses, supporting decentralized learning. This means that the TM most of the time will operate on outdated voting tallies. We evaluated the proposed parallelization across diverse learning tasks and it turns out that our decentralized TM learning algorithm copes well with working on outdated data, resulting in no significant loss in learning accuracy. Furthermore, we show that the proposed approach provides up to 50 times faster learning. Finally, learning time is almost constant for reasonable clause amounts (employing from 20 to 7 000 clauses on a Tesla V100 GPU). For sufficiently large clause numbers, computation time increases approximately proportionally. Our parallel and asynchronous architecture thus allows processing of massive datasets and operating with more clauses for higher accuracy.

[*]Equal contribution (The authors are ordered alphabetically by last name.) [1]Department of information and communication technology, Unviersity of Agder, Grimstad, Norway. Correspondence to: Ole-Christoffer Granmo <ole.granmo@uia.no>.

*Proceedings of the 38th International Conference on Machine Learning*, PMLR 139, 2021. Copyright 2021 by the author(s).

## 1. Introduction

Tsetlin machines (TMs) (Granmo, 2018) have recently demonstrated competitive results in terms of accuracy, memory footprint, energy, and learning speed on diverse benchmarks (image classification, regression, natural language understanding, and speech processing) (Berge et al., 2019; Yadav et al., 2021a; Abeyrathna et al., 2020; Granmo et al., 2019; Wheeldon et al., 2020; Abeyrathna et al., 2021; Lei et al., 2021). They use frequent pattern mining and resource allocation principles to extract common patterns in the data, rather than relying on minimizing output error, which is prone to overfitting. Unlike the intertwined nature of pattern representation in neural networks, a TM decomposes problems into self-contained patterns, expressed as conjunctive clauses in propositional logic (i.e., in the form **if** input $X$ **satisfies** condition $A$ **and not** condition $B$ **then** output $y = 1$). The clause outputs, in turn, are combined into a classification decision through summation and thresholding, akin to a logistic regression function, however, with binary weights and a unit step output function. Being based on the human-interpretable disjunctive normal form (Valiant, 1984), like Karnaugh maps (Karnaugh, 1953), a TM can map an exponential number of input feature value combinations to an appropriate output (Granmo, 2018).

**Recent progress on TMs** Recent research reports several distinct TM properties. The TM can be used in convolution, providing competitive performance on MNIST, Fashion-MNIST, and Kuzushiji-MNIST, in comparison with CNNs, K-Nearest Neighbor, Support Vector Machines, Random Forests, Gradient Boosting, BinaryConnect, Logistic Circuits and ResNet (Granmo et al., 2019). The TM has also achieved promising results in text classification (Berge et al., 2019), word sense disambiguation (Yadav et al., 2021b), novelty detection Bhattarai et al. (2021c;b), fake news detection (Bhattarai et al., 2021a), semantic relation analysis (Saha et al., 2020), and aspect-based sentiment analysis (Yadav et al., 2021a) using the conjunctive clauses to capture textual patterns. Recently, regression TMs compared favorably with Regression Trees, Random Forest Regression, and Support Vector Regression (Abeyrathna et al., 2020). The above TM approaches have further been enhanced by various techniques. By introducing real-valued clause weights, it turns out that the number of clauses can be reduced by up to $50\times$ without loss of accuracy (Phoulady et al., 2020). Also, the logical inference structure of TMs makes it possible to index the clauses on the features that falsify them, increasing inference- and learning speed by up to an order of magnitude (Gorji et al., 2020). Multi-granular clauses simplify the hyper-parameter search by eliminating the pattern specificity parameter (Gorji et al., 2019). In Abeyrathna et al. (2021), stochastic searching on the line automata (Oommen, 1997) learn integer clause weights, performing on-par or better than Random Forest, Gradient Boosting, Neural Additive Models, StructureBoost and Explainable Boosting Machines. Closed form formulas for both local and global TM interpretation, akin to SHAP, was proposed in Blakely & Granmo (2020). From a hardware perspective, energy usage can be traded off against accuracy by making inference deterministic (Abeyrathna et al., 2020). Additionally, Shafik et al. (2020) show that TMs can be fault-tolerant, completely masking stuck-at faults. Recent theoretical work proves convergence to the correct operator for "identity" and "not". It is further shown that arbitrarily rare patterns can be recognized, using a quasi-stationary Markov chain-based analysis. The work finally proves that when two patterns are incompatible, the most accurate pattern is selected (Zhang et al., 2020). Convergence for the "XOR" operator has also recently been proven in Jiao et al. (2021).

**Paper Contributions** In all of the above mentioned TM schemes, the clauses are learnt using Tsetlin automaton (TA)-teams (Tsetlin, 1961) that interact to build and integrate conjunctive clauses for decision-making. While producing accurate learning, this interaction creates a bottleneck that hinders parallelization. That is, the clauses must be evaluated and compared before feedback can be provided to the TAs.

In this paper, we first cover the basics of TMs in Section 2. Then, we propose a novel parallel and asynchronous architecture in Section 3, where every clause runs in its own thread for massive parallelism. We eliminate the above interaction bottleneck by introducing local voting tallies that keep track of the clause outputs, per training example. The local voting tallies detach the processing of each clause from the rest of the clauses, supporting decentralized learning. Thus, rather than processing training examples one-by-one as in the original TM, the clauses access the training examples simultaneously, updating themselves and the local voting tallies in parallel. In Section 4, we investigate the properties of the new architecture empirically on regression, novelty detection, semantic relation analysis and word sense disambiguation. We show that our decentralized TM architecture copes well with working on outdated data, with no measurable loss in learning accuracy. We further investigate how processing time scales with the number of clauses, uncovering almost constant-time processing over reasonable clause amounts. Finally, in Section 5, we conclude with pointers to future work, including architectures for grid-computing and heterogeneous systems spanning the cloud and the edge.

The main contributions of the proposed architecture can be summarized as follows:

- Learning time is made almost *constant* for reasonable clause amounts (employing from 20 to 7 000 clauses on a Tesla V100 GPU).
- For sufficiently large clause numbers, computation time increases approximately proportionally to the increase in number of clauses.
- The architecture copes remarkably with working on outdated data, resulting in no significant loss in learning accuracy across diverse learning tasks (regression, novelty detection, semantic relation analysis, and word sense disambiguation).

Our parallel and asynchronous architecture thus allows processing of more massive data sets and operating with more clauses for higher accuracy, significantly increasing the impact of logic-based machine learning.

## 2. Tsetlin Machine Basics

### 2.1. Classification

A TM takes a vector $X = [x_1, \ldots, x_o]$ of $o$ Boolean features as input, to be classified into one of two classes, $y = 0$ or $y = 1$. These features are then converted into a set of literals that consists of the features themselves as well as their negated counterparts: $L = \{x_1, \ldots, x_o, \neg x_1, \ldots, \neg x_o\}$.

If there are $m$ classes and $n$ sub-patterns per class, a TM employs $m \times n$ conjunctive clauses to represent the sub-patterns. For a given class[1], we index its clauses by $j$, $1 \leq j \leq n$, each clause being a conjunction of literals:

$$C_j(X) = \left(\bigwedge_{l_k \in L_j} l_k\right) \wedge \left(\bigwedge_{l_k \in \bar{L}_j} l_k\right). \qquad (1)$$

Here, $l_k, 1 \leq k \leq 2o$, is a feature or its negation. Further, $L_j$ is a subset of the unnegated features from the literal set $L$, whereas $\bar{L}_j$ is a subset of the negated features in $L$. For example, the particular clause $C_j(X) = x_1 \wedge x_2$ consists of the literals $L_j = \{x_1, x_2\}$, $\bar{L}_j = \emptyset$, and outputs 1 if $x_1 = x_2 = 1$.

[1]Without loss of generality, we consider only one of the classes, thus simplifying notation. Any TM class is modelled and processed in the same way.

The number of clauses $n$ assigned to each class is user-configurable. The clauses with odd indexes are assigned positive polarity and the clauses with even indexes are assigned negative polarity. The clause outputs are combined into a classification decision through summation and thresholding using the unit step function $u(v) = 1$ **if** $v \geq 0$ **else** $0$:

$$\hat{y} = u\left(\Sigma_{j=1,3,\ldots}^{n-1} C_j(X) - \Sigma_{i=2,4,\ldots}^{n} C_j(X)\right) \qquad (2)$$

Namely, classification is performed based on a majority vote, with the positive clauses voting for $y = 1$ and the negative for $y = 0$.

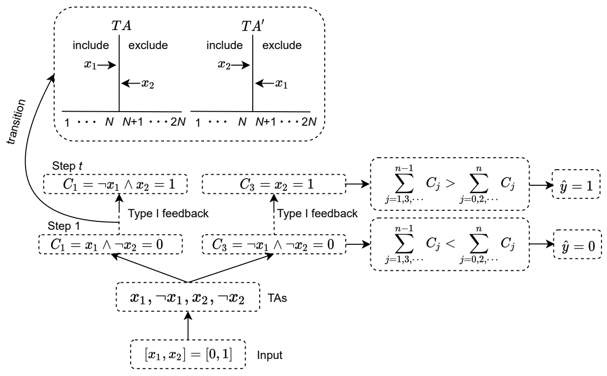
$TA$
include
exclude
$x_1$
$x_2$
$TA'$
include
exclude
$x_2$
$x_1$
1 ⋯ N N+1 ⋯ 2N
1 ⋯ N N+1 ⋯ 2N
transition
Step $t$
$C_1 = \neg x_1 \land x_2 = 1$
$C_3 = x_2 = 1$
$\sum_{j=1,3,\cdots}^{n-1} C_j > \sum_{j=0,2,\cdots}^{n} C_j$
$\hat{y} = 1$
Type I feedback
Type I feedback
Step 1
$C_1 = x_1 \land \neg x_2 = 0$
$C_3 = \neg x_1 \land \neg x_2 = 0$
$\sum_{j=1,3,\cdots}^{n-1} C_j < \sum_{j=0,2,\cdots}^{n} C_j$
$\hat{y} = 0$
$x_1, \neg x_1, x_2, \neg x_2$
TAs
$[x_1, x_2] = [0, 1]$
Input

*Figure 1.* TM learning dynamics for an XOR-gate training sample, with input $(x_1 = 0, x_2 = 1)$ and output target $y = 1$.

## 2.2. Learning

TM learning is illustrated in Fig. 1. As shown, a clause $C_j(X)$ is composed by a team of TAs. Each TA has $2N$ states and decides to *Include* (from state 1 to $N$) or *Exclude* (from state $N+1$ to $2N$ ) a specific literal $l_k$ in the clause. In the figure, TA refers to the TAs that control the original form of a feature ($x_1$ and $x_2$) while TA' refers to those controlling negated features ($\neg x_1$ and $\neg x_2$). A TA updates its state based on the feedback it receives in the form of Reward, Inaction, and Penalty (illustrated by the features moving in a given direction in the TA-part of the figure). There are two types of feedback associated with TM learning: Type I feedback and Type II feedback, which are shown in Table 1 and Table 2, respectively.

**Type I feedback** is given stochastically to clauses with odd indexes when $y = 1$ and to clauses with even indexes when $y = 0$. Each clause, in turn, reinforces its TAs based on: (1) its output $C_j(X)$; (2) the action of the TA – *Include* or *Exclude*; and (3) the value of the literal $l_k$ assigned to the TA. As shown in Table 1, two rules govern Type I 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[2] (triggered with high probability) and makes the clause remember and refine the pattern it recognizes in $X$.
- *Include* is penalized and *Exclude* is rewarded with probability $\frac{1}{s}$ **if** $C_j(X) = 0$ **or** $l_k = 0$. This reinforcement is weak (triggered with low probability) and coarsens infrequent patterns, making them frequent.

| Input | Clause | 1 | | 0 | |
|---|---|---|---|---|---|
| | Literal | 1 | 0 | 1 | 0 |
| 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 | $\frac{1}{s}$ | $\frac{1}{s}$ |
| 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 1.* Type I Feedback

| Input | Clause | 1 | | 0 | |
|---|---|---|---|---|---|
| | Literal | 1 | 0 | 1 | 0 |
| Include Literal | P(Reward) | 0 | NA | 0 | 0 |
| | P(Inaction) | 1.0 | NA | 1.0 | 1.0 |
| | P(Penalty) | 0 | NA | 0 | 0 |
| Exclude Literal | P(Reward) | 0 | 0 | 0 | 0 |
| | P(Inaction) | 1.0 | 0 | 1.0 | 1.0 |
| | P(Penalty) | 0 | 1.0 | 0 | 0 |

*Table 2.* Type II Feedback

Above, parameter $s$ controls pattern frequency.

**Type II feedback** is given stochastically to clauses with odd indexes when $y = 0$ and to clauses with even indexes when $y = 1$. As captured by Table 2, it penalizes *Exclude* with probability 1 **if** $C_j(X) = 1$ **and** $l_k = 0$. Thus, this feedback produces literals for discriminating between $y = 0$ and $y = 1$.

The "state" is actualized in the form of a simple counter per Tsetlin Automaton. In practice, reinforcing Include (penalizing Exclude or rewarding Include) is done by increasing the counter, while reinforcing exclude (penalizing Include or rewarding Exclude) is performed by decreasing the counter.

As an example of learning, let us consider a dataset with XOR-gate sub-patterns. In particular, consider the input $(x_1 = 0, x_2 = 1)$ and target output $y = 1$, to visualize the learning process (cf. Fig. 1). We further assume that we have $n = 4$ clauses per class. Among the 4 clauses, the clauses

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

$C_1$ and $C_3$ vote for $y = 1$ and the clauses $C_0$ and $C_2$ vote for $y = 0$. For clarity, let us only consider how $C_1$ and $C_3$ learn a sub-pattern from the given sample of XOR-gate input and output. At Step 1 in the figure, the clauses have not yet learnt the pattern for the given sample. This leads to the wrong class prediction ($\hat{y} = 0$), thereby triggering Type I feedback for the corresponding literals. Looking up clause output $C_1(X) = 0$ and literal value $x_1 = 0$ in Table 1, we note that the TA controlling $x_1$ receives either Inaction or Penalty feedback for including $x_1$ in $C_1$, with probability $\frac{s-1}{s}$ and $\frac{1}{s}$, respectively. After receiving several penalties, with high probability, the TA changes its state to selecting *Exclude*. Accordingly, literal $x_1$ gets removed from the clause $C_1$. On the other hand, the TA that has excluded literal $\neg x_1$ from $C_1$ also obtains penalties, and eventually switches to the *Include* side of its state space. The combined outcome of these updates are shown in Step $t$ for $C_1$. Similarly, the TA that has included literal $\neg x_2$ in clause $C_1$ receives Inaction or Penalty feedback with probability $\frac{s-1}{s}$ and $\frac{1}{s}$, respectively. After obtaining multiple penalties, with high probability, $\neg x_2$ becomes excluded from $C_1$. Simultaneously, the TA that controls $x_2$ ends up in the *Include* state, as also shown in Step $t$. At this point, both clause $C_1$ and $C_3$ outputs 1 for the given input, correctly predicting the output $\hat{y} = 1$.

**Resource allocation** dynamics ensure that clauses distribute themselves across the frequent patterns, rather than missing some and over-concentrating on others. That is, for any input $X$, the probability of reinforcing a clause gradually drops to zero as the clause output sum

$$v = \Sigma_{j=1,3,\ldots}^{n-1} C_j(X) - \Sigma_{i=2,4,\ldots}^{n} C_j(X) \tag{3}$$

approaches a user-set target $T$ for $y = 1$ (and $-T$ for $y = 0$). If a clause is not reinforced, it does not give feedback to its TAs, and these are thus left unchanged. In the extreme, when the voting sum $v$ equals or exceeds the target $T$ (the TM has successfully recognized the input $X$), no clauses are reinforced. They are then free to learn new patterns, naturally balancing the pattern representation resources (Granmo, 2018).

## 3. Parallel and Asynchronous Architecture

Even though CPUs have been traditionally geared to handle high workloads, they are more suited for sequential processing and their performance is still dependant on the limited number of cores available. In contrast, since GPUs are primarily designed for graphical applications by employing many small processing elements, they offer a large degree of parallelism (Owens et al., 2007). As a result, a growing body of research has been focused on performing general purpose GPU computation or GPGPU. For efficient use of GPU power, it is critical for the algorithm to expose a large amount of fine-grained parallelism (Jiang & Snir, 2005; Satish et al., 2009).

While the voting step of the TM (cf. Eq. 3) hinders parallelization, the remainder of the TM architecture is natively parallel. In this section, we introduce our decentralized inference scheme and the accompanying architecture that makes it possible to have parallel asynchronous learning and classification, resolving the voting bottleneck by using local voting tallies.

### 3.1. Voting Tally

A voting tally that tracks the aggregated output of the clauses for each training example is central to our scheme. In a standard TM, each training example $(X_i, y_i), 1 \leq i \leq M$, is processed by first evaluating the clauses on $X_i$ and then obtaining the majority vote $v$ from Eq. (3). Here $M$ is the total number of examples. The majority vote $v$ is then compared with the summation target $T$ when $y = 1$ and $-T$ when $y = 0$, to produce the feedback to the TAs of each clause, explained in the previous section.

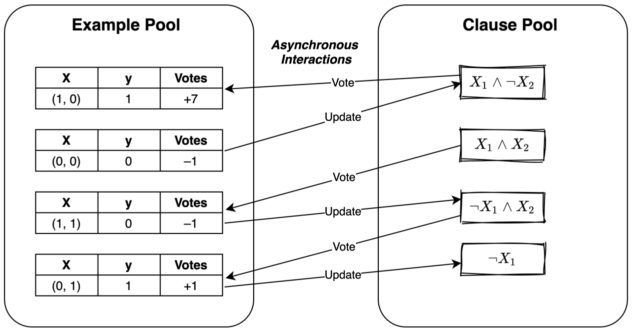

*Figure 2.* Parallel Tsetlin machine architecture.

As illustrated in Fig. 2, to decouple the clauses, we now assume that the particular majority vote of example $X_i$ has been pre-calculated, meaning that each training example becomes a triple $(X_i, y_i, v_i)$, where $v_i$ is the pre-calculated majority vote. With $v_i$ in place, the calculation performed in Eq. (3) can be skipped, and we can go directly to give Type I or Type II feedback to any clause $C_j$, without considering the other clauses. This opens up for decentralized learning of the clauses, facilitating native parallelization at all inference and learning steps. The drawback of this scheme, however, is that any time the composition of a clause changes after receiving feedback, all voting aggregates $v_i, 1 \leq i \leq M$, becomes outdated. Accordingly, the standard learning scheme for updating clauses must be replaced.

### 3.2. Decentralized Clause Learning

Our decentralized learning scheme is captured by Algorithm 1. As shown, each clause is trained independently of the other clauses. That is, each clause proceeds with training

**Algorithm 1** Decentralized updating of clause

```
**Input:** Example pool $P$, clause $C_j$, positive polarity indicator $p_j \in \{0,1\}$, batch size $b \in [1,\infty)$, voting target $T \in [1,\infty)$, pattern specificity $s \in [1,\infty)$.
**Procedure:** UpdateClause : $C_j, p_j, P, b, T, s$.
for $i = 1$ to $b$ do
    $(X_i, y_i, v_i) \leftarrow \mathrm{ObtainTrainingExample}(P)$
    $v_i^c \leftarrow$ clip $(v_i, -T, T)$
    $e = T - v_i^c$ if $y_i = 1$ else $T + v_i^c$
    if rand() $\leq \frac{e}{2T}$ then
        if $y_i$ xor $p_j$ then
            TypeIIFeedback($X_i, C_j$)
        else
            TypeIFeedback($X_i, C_j, s$)
        end if
        $o_{ij} \leftarrow C_j(X_i)$
        $o_{ij}^* \leftarrow \mathrm{ObtainPreviousClauseOutput}(i,j)$
        if $o_{ij} \neq o_{ij}^*$ then
            $\mathrm{AtomicAdd}(v_i, o_{ij} - o_{ij}^*)$
            $\mathrm{StorePreviousClauseOutput}(i, j, o_{ij})$
        end if
    end if
end for
```

without taking other clauses into consideration. Algorithm 1 thus supports native parallelization because each clause now can run independently in its own thread.

Notice further how the clause in focus first obtains a reference to the next training example $(X_i, y_i, v_i)$ to process, including the pre-recorded voting sum $v_i$ (Line 3). This example is retrieved from an example pool $P$, which is the storage of the training examples (centralized or decentralized).

The error of the pre-recorded voting sum $v_i$ is then calculated based on the voting target $T$ (Line 5). The error, in turn, decides the probability of updating the clause. The updating rule is the standard Type I and Type II TM feedback, governed by the polarity $p_j$ of the clause and the specificity hyper-parameter $s$ (Lines 6-11).

The moment clause $C_j$ is updated, all recorded voting sums in the example pool $P$ are potentially outdated. This is because $C_j$ now captures a different pattern. Thus, to keep all of the voting sums $v_i$ in $P$ consistent with $C_j$, $C_j$ should ideally have been re-evaluated on all of the examples in $P$.

To partially remedy for outdated voting aggregates, the clause only updates the *current* voting sum $v_i$. This happens when the calculated clause output $o_{ij}$ is different from the previously calculated clause output $o_{ij}^*$ (Lines 12-17). Note that the previously recorded output $o_{ij}^*$ is a single bit that is stored locally together with the clause. In this manner, the algorithm provides *eventual consistency*. That is, if the clauses stop changing, all the voting sums eventually become correct.

A point to note here, is that there is no way to guarantee that the clauses in the parallel version will sum up to the exact same number as in the sequential version. This is because the updating of clauses is asynchronous and the vote sums are not updated immediately when a clause changes. We use the term *eventual consistency* in this case, only to refers to the fact that if the clauses stop changing, eventually, the tallied voting sums stop being outdated (i.e. they become the exact sum of the clause outputs). Although not analytically proven, experimental results show that the two versions provide consistent final accuracy results, after clause summation and thresholding.

Employing the above algorithm, the clauses access the training examples simultaneously, updating themselves and the local voting tallies in parallel. There is no synchronization among the clause threads, apart from atomic adds to the local voting tallies (Line 15).

## 4. Empirical Results

In this section, we investigate how our new approach of TM learning scales, including effects on training time and accuracy. We employ seven different data sets that represent diverse learning tasks, including regression, novelty detection, sentiment analysis, semantic relation analysis, and word sense disambiguation. The data sets are of various sizes, spanning from 300 to $100,000$ examples, 2 to 20 classes, and 6 to $102,176$ features. We have striven to recreate TM experiments reported by various researchers, including their hyper-parameter settings. For comparison of performance, we contrast with fast single-core TM implementations[3] both with and without clause indexing (Gorji et al., 2020). Our proposed architecture is implemented in CUDA and runs on a Tesla V100 GPU (grid size 208 and block size 128). The standard implementations run on an Intel Xeon Platinum 8168 CPU at 2.70 GHz.

We summarize the obtained performance metrics in Table 3. For greater reproducibility, each experiment is repeated five times and the average accuracy and standard deviation are reported. We also report how much faster the CUDA TM executes compared with the indexed version.

Though the focus in this paper is to highlight the speed-ups achieved via parallelization of the Tsetlin Machine architecture, for the sake of completeness, we compare the same with a baseline set using ThunderSVM, which is an established GPU implementation[4] of SVM (Wen et al., 2018)

[3]Retrieved from https://github.com/cair/pyTsetlinMachine.

[4]Google's Colab is used to run the two CUDA models in the table for fair comparison.

| Dataset | TM indexed | | TM non-indexed | | TM CUDA | | Speed up |
|---|---|---|---|---|---|---|---|
| | Acc | F1 | Acc | F1 | Acc | F1 | |
| BBC Sports | 85.08 ± 1.75 | 85.58 ± 1.69 | 87.36 ± 1.91 | 88.02 ± 1.47 | 84.64 ±2.20 | 86.13 ± 2.24 | 38.9× |
| 20 Newsgroup | 79.37 ± 0.25 | 80.38 ± 0.92 | 82.33 ± 0.28 | 82.89 ± 0.34 | 79.00 ± 0.46 | 78.93 ± 0.44 | 49.3× |
| SEMEVAL | 91.9 ± 0.16 | 75.29 ± 0.25 | 92.51 ± 0.03 | 77.48 ± 1.46 | 92.02 ± 0.54 | 76.27 ± 0.53 | 1.7× |
| IMDB | 88.42 ± 2.05 | 88.39 ± 2.16 | 88.2 ± 3.14 | 88.13 ± 3.44 | 89.92 ± 0.23 | 88.90 ± 0.24 | 34.6× |
| JAVA (WSD) | 97.03 ± 0.02 | 96.93 ± 0.02 | 97.50 ± 0.02 | 97.40 ± 0.02 | 97.53 ± 0.02 | 97.42 ± 0.01 | 6.0× |
| Apple (WSD) | 92.65 ± 0.02 | 92.20 ± 0.02 | 92.46 ± 0.01 | 91.82 ± 0.02 | 95.01 ± 0.01 | 94.68 ± 0.01 | 9.7× |

*Table 3.* Performance on multiple data sets. Mean and standard deviation are calculated over 5 independent runs. Speed up is calculated as how many times faster is average execution time on CUDA implementation than on Indexed implementation.

as shown in Fig. 4. It can be seen that parallelization in TM is significantly faster in compared to ThunderSVM in all selected datasets. Note that our parallelization solution leverages novel TM properties, allowing each TM clause to learn independently in its own thread.

## 4.1. Regression

We first investigate performance with regression Tsetlin machines (RTMs) using two datasets: Bike Sharing and BlogFeedback.

The Bike Sharing dataset contains a 2-year usage log of the bike sharing system Captial Bike Sharing (CBS) at Washington, D.C., USA. The total number of rental bikes (casual and registered), is predicted based on other relevant features such as time of day, day of week, season, and weather. In total, 16 independent features are employed to predict the number of rental bikes. Overall, the dataset contains 17389 examples. More details of the Bike Sharing dataset can be found in Fanaee-T & Gama (2014).

The BlogFeedback dataset considers predicting the number of blog posts comments for the upcoming 24 hours. The data has been gathered from January, 2010 to March, 2012. In total, BlogFeedback contains 60021 data samples, each with 281 features. In this study, we employ the first 20 000 data samples. For both of the datasets, we use 80% of the samples for training and the rest for evaluation.

We first study the impact of the number of clauses on prediction error, measured by Mean Absolute Error (MAE). As illustrated in Fig. 3 for Bike Sharing, increasing the number of clauses (#clauses in x-axis) decreases the error by allowing the RTM to capture more detailed sub-patterns.

A larger number of clauses results in increased computation time. Fig. 4 captures how execution time increases with the number of clauses for the three different implementations. For the non-indexed RTM, each doubling of the number of clauses also doubles execution time. However, the indexed RTM is less affected, and is slightly faster than the CUDA implementation when less than 300 clauses are utilized. With more than 300 clauses, the CUDA implementation is superior, with no significant increase in execution time as the number of clauses increases. For instance, using 1280 clauses, the CUDA implementation is roughly 3.64 times faster than the indexed version. This can be explained by the large number of threads available to the GPU and the asynchronous operation of the new architecture.

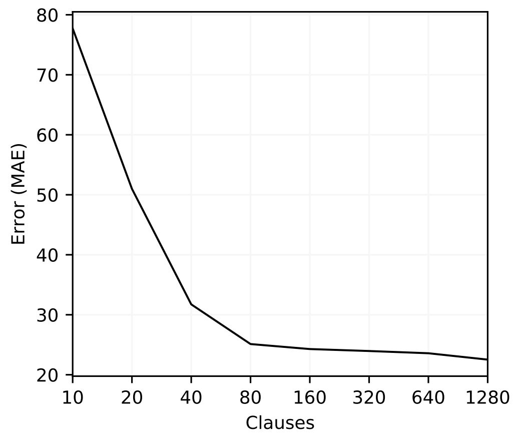

*Figure 3.* MAE vs. #clauses on Bike Sharing.

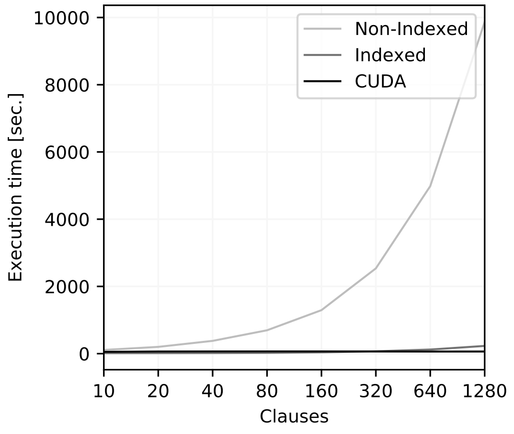

*Figure 4.* Execution time vs. #clauses on Bike Sharing.

Looking at how MAE and execution time vary over the

| CUDA MODEL | BBC SPORTS | 20 NG | SEMEVAL | IMDB | JAVA(WSD) | APPLE(WSD) | REGR1 | REGR2 |
|---|---|---|---|---|---|---|---|---|
| THUNDERSVM | 0.23S | 6.95S | 7.42S | 3.89S | 1.31S | 0.39S | 6.24S | 59.94S |
| TM | 0.13S | 3.61S | 4.27S | 3.36S | 0.74S | 0.28S | 0.52S | 4.92S |

*Table 4.* Comparison of training time between ThunderSVM and TM. The run time is an average of 5 experiments.

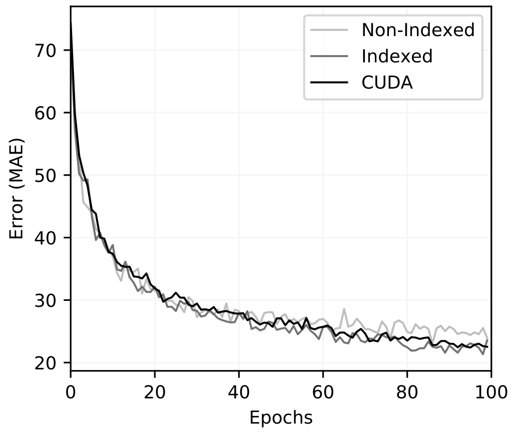

*Figure 5.* MAE over epochs on Bike Sharing.

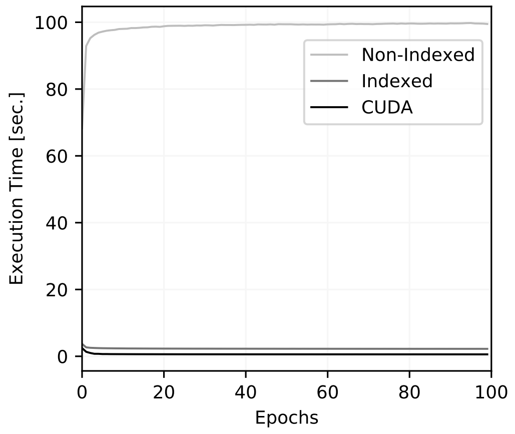

*Figure 6.* Per Epoch Execution time over epochs on Bike Sharing.

training epochs for Bike Sharing (Fig. 5 and Fig. 6, respectively), we observe that MAE falls systematically across the epochs, while the execution time remains stable (employing $T = 1280$, $s = 1.5$, $n = 1280$). Execution on BlogFeedback exhibits similar behavior. Finally, the mean value of the MAEs for each method is similar (Table 5) across 5 independent runs, indicating no significant difference in learning accuracy.

MAE for Regression analysis using comparable contemporary methods are as follows: Bike Sharing (Regr1): TM: 23.9; ANN: 27.4; SVR: 22.6. BlockFeedback (Regr2): TM: 3.88; ANN: 7.2; SVR: 6.0.

## 4.2. Novelty Detection

Novelty detection is another important machine learning task. Most supervised classification approaches assume a closed world, counting on all classes being present in the data at training time. This assumption can lead to unpredictable behaviour during operation, whenever novel, previously unseen, classes appear. We here investigate TM-based novelty detection, as proposed in Bhattarai et al. (2021c;b), using two datasets: 20 Newsgroup and BBC Sports. In brief, we use the class voting sums (Section 2) as features measuring novelty. We then employ a Multilayer perceptron (MLP) for novelty detection that adopts the class voting sums as input.

The BBC sports dataset contains 737 documents from the BBC sport website, organized in five sports article categories and collected from 2004 to 2005. Overall, the dataset encompasses a total of $4,613$ terms. For novelty classification, we designate the classes “Cricket” and “Football” as known and “Rugby” as novel. We train on the known classes, which runs for 100 epochs with $5,000$ clauses, threshold $T$ of 100, and sensitivity $s$ of 15.0. The training times for both indexed and non-indexed TMs are high compared with that of CUDA TM, which is around 39 times faster. The 20 Newsgroup dataset contains 18 828 documents with 20 classes. The classes “comp.graphics” and “talk.politics.guns” are designated as known, and “rec.sport.baseball” is considered novel. We train the TM for 100 epochs with a target $T$ of 500, 10 000 clauses and sensitivity $s = 25.0$. The CUDA TM implementation is here about 49 times faster than the other versions.

To assess scalability, we record the execution time of both the indexed and the CUDA TM while increasing the number of clauses (Fig. 7). For the indexed TM, the execution time increases almost proportionally with the number of clauses, but no such effect is noticeable for the CUDA TM.

The novelty scores generated by the TM are passed to a Multilayer Perceptron with hidden layer sizes (100, 30) and RELU activation functions, trained using stochastic gradient descent. As shown in Table 3, for both datasets, the non-indexed TM slightly outperforms the other TM versions, while the indexed and CUDA TMs have similar accuracy. These differences can be explained by the random variation of TM learning (i.e., the high standard deviations reported in Table 3).

Bhattarai et al. (2021c) also illustrates the following compar-

| Datasets | TM indexed | TM non-indexed | TM CUDA | Speed up |
|---|---|---|---|---|
| Bike Sharing | 23.5±0.04 | 22.5±0.12 | 23.9±0.08 | 156.4× |
| BlogFeedback. | 3.91±0.00 | 3.74±0.03 | 3.88±0.02 | 211.6× |

*Table 5.* MAE with confidence interval, and Speed up on two regression datasets, calculated over 5 independent runs

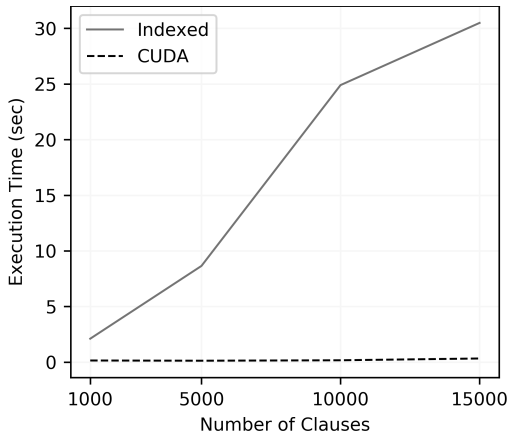

*Figure 7.* Execution time vs. #clauses on BBC Sports.

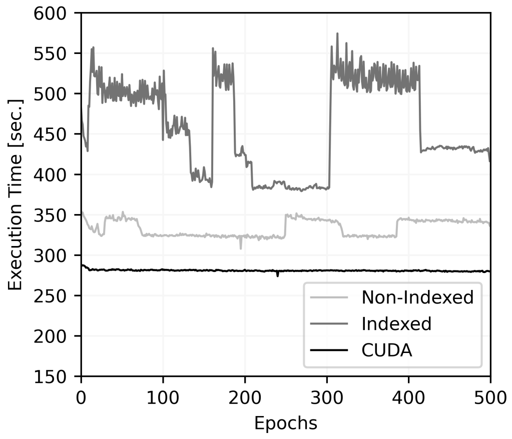

*Figure 8.* Per Epoch Execution time over epochs on SEMEVAL.

ison in terms of accuracy with other relevant works: BBC Sports, TM: 89.47%; One-class SVM: 83.53%; Average KNN: 55.54%. For 20Newsgroup, TM: 82.51%; One-class SVM: 83.70%; Average KNN: 81%.

## 4.3. Sentiment and Semantic Analysis

We adopt the *SemEval 2010 Semantic Relations* (Hendrickx et al., 2009) and the *ACL Internet Movie Database (IMDb)* (Maas et al., 2011) datasets to explore the performance of the TM implementations for a large number of sparse features, as proposed in Saha et al. (2020).

The SEMEVAL dataset focuses on identifying semantic relations in text. The dataset has 10 717 examples, and we consider each to be annotated to contain either the relation Cause-Effect or not. The presence of an unambiguous causal connective is indicative of a sentence being a causal sentence (Xuelan & Kennedy, 1992). For each TM, we use 40 clauses per class to identify this characteristic of causal texts. The IMDb dataset contains 50 000 highly polar movie reviews, which are either positive or negative. Due to the large variety and combination of possibly distinguishing features, we assign 7 000 clauses to each class. For both datasets we use unigrams and bigrams as features.

As noted in Table 3, the accuracy obtained by the CPU (non-indexed) and the CUDA implementations are comparable on the SEMEVAL dataset, while the indexed TM performs slightly poorer. However, the execution time is much lower for the CUDA version than the other two (Fig. 8). This is

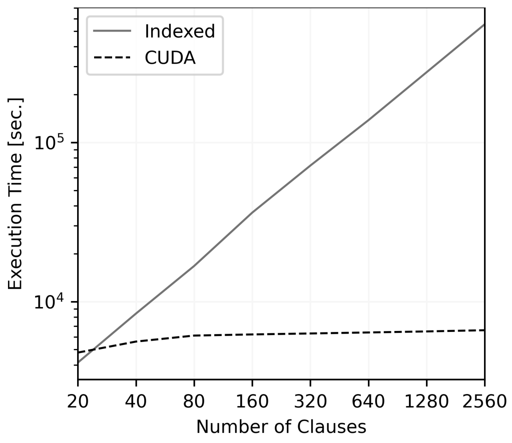

*Figure 9.* Execution time vs. #clauses on SEMEVAL.

further shown in Fig. 9. Clearly, the CPU-based TM with indexing takes an increasing amount of time to execute as the number of clauses grows, but no such effect is observed with CUDA TM. More specifically, the execution time increases only by 40% when the number of clauses goes from 20 to 2 560.

With the IMDB dataset, the CUDA version performs better in terms of accuracy, with less variance compared to the CPU versions (Table 3). It exhibits similar behaviour as in the SEMEVAL dataset with respect to execution time over increasing number of epochs.

The following provides a comparison of accuracy, compiled from related literature: **Semantic Relation Classification. (SemEval).** TM: 92.6%; RandomForest: 88.76%; NaiveBayes: 87.57%; SVM: 92.3%; CNN-LSTM: 90.73%. **IMDB Sentiment Analysis.** TM: 90.5%; NBOW: 83.62%; CNN-BiLSTM: 89.54%; Tree-LSTM: 90.1%; Self-AT-LSTM: 90.34%; DistilBERT: 92.82%.

### 4.4. Word Sense Disambiguation

Word Sense Disambiguation (WSD) is a vital task in NLP (Navigli, 2009) that consists of distinguishing the meaning of homonyms – identically spelled words whose sense depends on the surrounding context words. We here perform a quantitative evaluation of the three TM implementations using a recent WSD evaluation framework (Loureiro et al., 2020) based on WordNet. We use a balanced dataset for coarse grained classification, focusing on two specific domains. The first dataset concerns the meaning of the word "Apple", which here has two senses: "apple_inc." (company) and "apple_apple" (fruit). The other dataset covers the word "JAVA", which has the two senses: "java_java" (geographical location) and "java_comp." (computer language). The Apple dataset has 2 984 samples split into training and testing samples of 1 784 and 1 200, respectively. The JAVA dataset has 5 655 samples split into 3 726 and 1 929 samples, for training and testing. For preprocessing, we filter the stop words and stem the words using the Porter Stemmer to reduce the effect of spelling mistakes or non-important variations of the same word. To build a vocabulary (the feature space), we select the 3 000 most frequent terms. The number of clauses, threshold, and specificity used are 300, 50, 5 respectively, for both datasets.

The accuracy and F1 score of non-indexed and indexed TMs is quite similar for the Apple dataset (Table 3). However, the CUDA TM outperforms both of them by a significant margin. In the case of JAVA dataset, the performance is comparable for all three, and CUDA TM is slightly better. The reader is requested to refer to Yadav et al. (2021b) for further details on TM-based WSD.

The following is a comparison of accuracy with respect to related literature in Loureiro et al. (2020) TM(Apple): 95.1%; ThunderSVM(Apple): 80.32%; FastTextBase-1NN(Apple): 96.3%; FastTextCommonCrawl-1NN(Apple): 97.8%; BERTbase(Apple): 99.0%. TM(JAVA): 97.53%; ThunderSVM(JAVA): 93.62%; FastTextBase-1NN(JAVA): 98.7%; FastTextCommonCrawl-1NN(JAVA): 99.6%; BERTbase(JAVA): 99.0%. These TM results are from our current paper. Another recent paper, however, shows that better hyperparameters can make the TM outperform FastTextBase-1NN with TM reaching 97.58% and 99.38% for Apple and JAVA, respectively (Yadav et al., 2021b).

### 4.5. Analysis of Results obtained from Experiments

The above results support our initial claims that the proposed Tsetlin Machine architecture results in learning times that are nearly constant for a reasonable number of clauses, and beyond a sufficiently large number, the computation time increases at approximately the same rate as the increase in number of clauses. The number of clauses to use highly depends on the dataset. In our case, using 7000 clauses provides high accuracy for all the datasets, simultaneously demonstrating almost constant training time overall. A further increase does not significantly improve accuracy. I.e., the number of clauses was decided empirically to provide a general benchmark. Employing 7000 clauses also allows us evaluate the capability of leveraging the 5120 cores on the GPU. Hence, considering both datasets and available cores, 7000 clauses are enough to demonstrate the almost constant time scaling, until exhausting the cores. However, we also extended the clauses to 15000 as shown in Fig. 7, and the execution time remained almost the same, thereby validating excellent exploitation of the available cores. TMs are further reported to use less memory than ANNs (Lei et al., 2020). Our scheme adds 1 bit per example per clause for tallying, enabling desynchronization.

## 5. Conclusions and Future Work

In this paper, we proposed a new approach to TM learning, to open up for massively parallel processing. Rather than processing training examples one-by-one as in the original TM, the clauses access the training examples simultaneously, updating themselves and local voting tallies in parallel. The local voting tallies allow us to detach the processing of each clause from the rest of the clauses, supporting decentralized learning. There is no synchronization among the clause threads, apart from atomic adds to the local voting tallies. Operating asynchronously, each team of TAs most of the time executes on partially calculated or outdated voting tallies.

Based on the numerical results, our main conclusion is that TM learning is very robust towards relatively severe distortions of communication and coordination among the clauses. Our results are thus compatible with the findings in Shafik et al. (2020), where it is shown that TM learning is inherently fault tolerant, completely masking stuck-at faults.

In our future work, we will investigate the robustness of TM learning further, which includes developing mechanisms for heterogeneous architectures and more loosely coupled systems, such as grid-computing.

## References

Abeyrathna, K. D., Granmo, O.-C., Shafik, R., Yakovlev, A., Wheeldon, A., Lei, J., and Goodwin, M. A Novel Multi-Step Finite-State Automaton for Arbitrarily Deterministic Tsetlin Machine Learning. In *Proceedings of the 40th International Conference on Innovative Techniques and Applications of Artificial Intelligence (SGAI), Cambridge, UK*. Springer International Publishing, 2020.

Abeyrathna, K. D., Granmo, O.-C., Zhang, X., Jiao, L., and Goodwin, M. The Regression Tsetlin Machine - A Novel Approach to Interpretable Non-Linear Regression. *Philosophical Transactions of the Royal Society A*, 378, 2020.

Abeyrathna, K. D., Granmo, O.-C., and Goodwin, M. Extending the Tsetlin Machine With Integer-Weighted Clauses for Increased Interpretability. *IEEE Access*, 9: 8233 – 8248, 2021.

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

Bhattarai, B., Granmo, O.-C., and Jiao, L. Explainable tsetlin machine framework for fake news detection with credibility score assessment, 2021a.

Bhattarai, B., Granmo, O.-C., and Jiao, L. Word-level human interpretable scoring mechanism for novel text detection using tsetlin machines, 2021b.

Bhattarai, B., Jiao, L., and Granmo, O.-C. Measuring the Novelty of Natural Language Text Using the Conjunctive Clauses of a Tsetlin Machine Text Classifier. In *13th International Conference on Agents and Artificial Intelligence (ICAART), Vienna , Austria*. INSTICC, 2021c.

Blakely, C. D. and Granmo, O.-C. Closed-Form Expressions for Global and Local Interpretation of Tsetlin Machines with Applications to Explaining High-Dimensional Data. *arXiv preprint arXiv:2007.13885*, 2020.

Fanaee-T, H. and Gama, J. Event labeling combining ensemble detectors and background knowledge. *Progress in Artificial Intelligence*, 2(2-3):113–127, 2014.

Gorji, S., Granmo, O. C., Glimsdal, S., Edwards, J., and Goodwin, M. Increasing the Inference and Learning Speed of Tsetlin Machines with Clause Indexing. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE), kitakyushu, Japan*. Springer, 2020.

Gorji, S. R., Granmo, O.-C., Phoulady, A., and Goodwin, M. A Tsetlin Machine with Multigranular Clauses. In *Proceedings of the Thirty-ninth International Conference on Innovative Techniques and Applications of Artificial Intelligence (SGAI), Cambridge, UK*, volume 11927. Springer International Publishing, 2019.

Granmo, O.-C. The Tsetlin Machine - A Game Theoretic Bandit Driven Approach to Optimal Pattern Recognition with Propositional Logic. *arXiv preprint arXiv:1804.01508*, 2018.

Granmo, O.-C., Glimsdal, S., Jiao, L., Goodwin, M., Omlin, C. W., and Berge, G. T. The Convolutional Tsetlin Machine. *arXiv preprint arXiv:1905.09688*, 2019.

Hendrickx, I., Kim, S. N., Kozareva, Z., Nakov, P., Ó Séaghdha, D., Padó, S., Pennacchiotti, M., Romano, L., and Szpakowicz, S. Semeval-2010 task 8: Multi-way classification of semantic relations between pairs of nominals. In *Proceedings of the Workshop on Semantic Evaluations: Recent Achievements and Future Directions, Boulder, Colorado*, pp. 94–99. ACL, 2009.

Jiang, C. and Snir, M. Automatic tuning matrix multiplication performance on graphics hardware. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05), Krasnoyarsk, Russia*, pp. 185–194. IEEE, 2005.

Jiao, L., Zhang, X., Granmo, O.-C., and Abeyrathna, K. D. On the Convergence of Tsetlin Machines for the XOR Operator. *arXiv preprint arXiv:2101.02547*, 2021.

Karnaugh, M. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5):593–599, 1953.

Lei, J., Wheeldon, A., Shafik, R., Yakovlev, A., and Granmo, O.-C. From arithmetic to logic based ai: A comparative analysis of neural networks and tsetlin machine. In *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 1–4. IEEE, 2020.

Lei, J., Rahman, T., Shafik, R., Wheeldon, A., Yakovlev, A., Granmo, O.-C., Kawsar, F., and Mathur, A. Low-Power Audio Keyword Spotting using Tsetlin Machines. *arXiv preprint arXiv:2101.11336*, 2021.

Loureiro, D., Rezaee, K., Pilehvar, M. T., and Camacho-Collados, J. Language models and word sense disambiguation: An overview and analysis, 2020.

Maas, A., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language* *technologies, Portland, Oregon, USA*, pp. 142–150. ACL, 2011.

Navigli, R. Word sense disambiguation: A survey. *ACM Computing Surveys*, 41:10:1–10:69, 2009.

Oommen, B. J. Stochastic searching on the line and its applications to parameter learning in nonlinear optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 27(4):733–739, 1997.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pp. 80–113. Wiley Online Library, 2007.

Phoulady, A., Granmo, O.-C., Gorji, S. R., and Phoulady, H. A. The Weighted Tsetlin Machine: Compressed Representations with Clause Weighting. In *Proceedings of the Ninth International Workshop on Statistical Relational AI (StarAI), New York, USA*, 2020.

Saha, R., Granmo, O.-C., and Goodwin, M. Mining Interpretable Rules for Sentiment and Semantic Relation Analysis using Tsetlin Machines. In *Proceedings of the 40th International Conference on Innovative Techniques and Applications of Artificial Intelligence (SGAI), Cambridge, UK*. Springer International Publishing, 2020.

Satish, N., Harris, M., and Garland, M. Designing efficient sorting algorithms for manycore gpus. In *2009 IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy*, pp. 1–10. IEEE, 2009.

Shafik, R., Wheeldon, A., and Yakovlev, A. Explainability and Dependability Analysis of Learning Automata based AI Hardware. In *IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS), Naples, Italy*. IEEE, 2020.

Tsetlin, M. L. On behaviour of finite automata in random medium. *Avtomat. i Telemekh*, 22(10):1345–1354, 1961.

Valiant, L. G. A Theory of the Learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

Wen, Z., Shi, J., Li, Q., He, B., and Chen, J. Thundersvm: A fast svm library on gpus and cpus. *The Journal of Machine Learning Research*, 19(1):797–801, 2018.

Wheeldon, A., Shafik, R., Rahman, T., Lei, J., Yakovlev, A., and Granmo, O.-C. Learning Automata based Energy-efficient AI Hardware Design for IoT. *Philosophical Transactions of the Royal Society A*, 2020.

Xuelan, F. and Kennedy, G. Expressing causation in written english. *RELC Journal*, 23(1):62–80, 1992.

Yadav, R. K., Jiao, L., Granmo, O.-C., and Goodwin, M. Human-Level Interpretable Learning for Aspect-Based Sentiment Analysis. In *Proceedings of AAAI, Vancouver, Canada*. AAAI, 2021a.

Yadav, R. K., Jiao, L., Granmo, O.-C., and Goodwin, M. Interpretability in Word Sense Disambiguation using Tsetlin Machine. In *13th International Conference on Agents and Artificial Intelligence (ICAART), Vienna, Austria*. INSTICC, 2021b.

Zhang, X., Jiao, L., Granmo, O.-C., and Goodwin, M. On the Convergence of Tsetlin Machines for the IDENTITY- and NOT Operators. *arXiv preprint arXiv:2007.14268*, 2020.