UiA University of Agder
Norway

A Scalable Architecture for Parallel Execution of the Tsetlin Machine

Anders Refsdal Olsen

SUPERVISOR
Prof. Ole-Christoffer Granmo

University of Agder, 2019
Faculty of Engineering and Science
Department of ICT

UiA
University of Agder
Master's thesis

Faculty of Engineering and Science
Department of ICT

Abstract

With the Tsetlin Machine recently released, much research has been done on its capabilities, with great success. However, the lack of tools available, and general knowledge of the Tsetlin Machine prevents it from being adopted by the industry. As a result, it is today mostly used in academic environments. To increase the general availability of the algorithm, this thesis introduces an introductory description to the algorithm and proposes an architecture that allows the use of multiple CPU threads and multiple GPUs to execute the algorithm in parallel. In addition, this thesis investigates several key aspects of the algorithm and how it could be improved, like introducing dynamic self-learning parameters and parallel reduction techniques for the Tsetlin Machine. The results show that the proposed architecture improves execution speed. Further, the Tsetlin Machine was able to adjust its own "s" parameter from a bad initial parameter towards, and finally converge with the known optimal value. The results from this thesis lay the foundation for creating powerful tools that allow for rapid development using the Tsetlin Machine in popular languages with high performance.

Preface

This thesis summarizes my work for the last two semester at the University of Agder. The task was proposed by both myself and Prof. Ole-Christoffer Granmo. The task proved to be quite challenging and I personally enjoyed the overwhelmingly amount of work that was required to complete this task. Most of all, I am happy that I got the opportunity to work on such a project, and I hope that my contribution is of any value to you as a reader.

During my thesis studies, I have been motivated and guided by the people around me. Primarily, I would like to extend my gratitude to my supervisor Prof. Ole-Christoffer Granmo. During this last year, I have learned more than any year of my stay at UiA. I wish you good luck with the Tsetlin Machine and I hope my work contributes to your research.

For writing this thesis, I have been motivated and received support from several people. This include, Ph.D. Raheleh Jafari who has helped me with writing this thesis and given me valuable, if not vital input on the report.

I have also been enjoying the company of several skilled researchers in the "Tsetlin Machine community" of CAIR. In addition to those already named, I appreciate all the insightful conversations with Ph.D. Candidate Saeed Rahimi Gorji and Ph.D. Candidate Darshana Abeyrathna.

List of Figures

2.1 A single layer neural network. 10
2.2 A small deep neural network (DNN). 11
2.3 One Tsetlin Automaton and how the states describes what action/output it will yield. 12
2.4 A small collective of 5 Automata, where the max state is 4, ranging from 1 to 4 inclusive. The output of each Automata is highlighted in different colors; green represents 1; red represents 0. 14
2.5 A simple GPU Kernel that squares all values in a matrix and stores them in a new matrix. 16

3.1 The Classical Multi-class Tsetlin Machine. 18
3.2 The conversion process of a Tsetlin Machine model to a state model. The *Output* function that is utilized is Equation 2.2. In this example, the $state_{\max}$ is set to the value of 200. 22
3.3 A sample with 6 given features, is expanded to a new sample with 12 features. 23
3.4 A sample and a state model that uses the IMPLY operation on each of the elements to create conjunctive clauses consisting of logical AND operations. 24

4.1 2 2D-matrices represented using one 1D array. 33
4.2 The flow of execution with the CPU training process. 35
4.3 The training process in detail for one class on CPU. 36
4.4 The flow of execution with the CPU evaluation process. 38
4.5 Streams tasked with handling one class each on one GPU. 40
4.6 Streams tasked with handling one class each on multiple GPU. . . 41
4.7 The flow of execution with the GPU training process. 45
4.8 CPU and GPU together train one single class in deeper detail. . . 47
4.9 CPU and GPU together evaluate one single class in further detail. 48
4.10 The execution time of an entire epoch for each of the configurations in experiment 1. 52

4.11 The execution time of an entire epoch for each of the configurations in experiment 2. 53

5.1 The current implementation of the Tsetlin Machine uses one CPU to count the output from each Clause in a Class. 56
5.2 Multiple threads calculate a partial sum using a stridden (stride) loop through the array of Clause outputs. Finally, the output is stored in shared memory after being altered by the polarity of the Clauses. 57
5.3 An example of how to reduce the partial sums from the first step using 8 threads and 16 sums. 58
5.4 A comparison of the average results from CPU and GPU reduction from Table 5.1. 61

6.1 An example of how the s changes during training using SSL using an initial value of 5.5 and a d of 0.5. 64
6.2 Plot from each of the experiments from the dynamic s value. . . . 67
6.3 The s changes during the epochs for experiment 1. 68
6.4 The s changes during the epochs for experiment 2. 69
6.5 The s changes during the epochs for experiment 3. 70
6.6 The s changes during the epochs for experiment 4. 71

List of Tables

2.1 Truth table for AND gate/operation 15
2.2 Truth table for IMPLY gate/operation 15

3.1 Shows how documents can be converted into vectors that acts as input data for a Tsetlin Machine model. 19
3.2 Probability table for Type 1 Feedback. [16] 27
3.3 Probability table for Type 2 Feedback. [16] 28

4.1 An example of how the distribution of Streams and GPUs look like on a system with 3 GPUs. 43
4.2 The configuration of the datasets used in the experiments for the proposed architecture. 49
4.3 The configuration of the models used in the experiments for the proposed architecture. 50
4.4 The training parameters used in the experiments for the proposed architecture. 50
4.5 The results when measuring the execution time from experiment 1 with a multi-threaded CPU implementation, 1 GPU and 2 GPUs. 51
4.6 The results when measuring the execution time from experiment 2 with a multi-threaded CPU implementation, 1 GPU, 2 GPUs, and 3 GPUs. 53

5.1 Average execution time from running the reduction experiment 50 000 times on both CPU and GPU. Time is represented in seconds. 60

Table of Contents

Abstract iii

Preface v

List of Figures viii

List of Tables ix

1 Introduction 3
- 1.1 Motivation . . . 3
- 1.2 Thesis definition . . . 4
  - 1.2.1 Thesis Goals . . . 4
  - 1.2.2 Hypotheses . . . 5
  - 1.2.3 Scope & Limitations . . . 5
- 1.3 Research Methodology . . . 6
- 1.4 Contributions . . . 7
- 1.5 Thesis outline . . . 7

2 Background 9
- 2.1 Neural Networks . . . 9
- 2.2 Learning Automaton . . . 12
  - 2.2.1 Input data . . . 12
  - 2.2.2 Training process . . . 13
  - 2.2.3 Output . . . 13
- 2.3 Collective of Learning Automatas . . . 14
- 2.4 Logical gates . . . 15
- 2.5 Kernels . . . 16

3 Introduction to the Tsetlin Machine 17
- 3.1 Multi-class Tsetlin Machine . . . 17
- 3.2 Structure . . . 19

3.2.1 Input data . . . 19
3.2.2 Model . . . 20
3.3 Training . . . 21
3.3.1 Conversion . . . 21
3.3.2 Evaluate . . . 23
3.3.3 Counting votes . . . 25
3.3.4 Calculate feedback . . . 26
3.3.5 Perform feedback . . . 27
3.4 Conclusion . . . 29

4 Proposed Architecture **31**
4.1 Model . . . 31
4.2 CPU . . . 33
4.2.1 Training . . . 34
4.2.2 Evaluation . . . 38
4.3 GPU . . . 39
4.3.1 Distribution of work . . . 40
4.3.2 Kernels . . . 41
4.3.3 Multiple GPUs . . . 42
4.3.4 Training . . . 43
4.3.5 Evaluation . . . 47
4.4 Experiments . . . 49
4.4.1 Datasets . . . 49
4.4.2 Models . . . 49
4.4.3 Training parameters . . . 50
4.4.4 Metrics and parameters . . . 50
4.5 Results . . . 51
4.5.1 Experiment 1 . . . 51
4.5.2 Experiment 2 . . . 52
4.6 Conclusion . . . 54

5 CPU and GPU reduction **55**
5.1 The problem with regular counting . . . 55
5.2 Proposed solution . . . 56
5.3 Experiment . . . 59
5.4 Results . . . 59
5.5 Conclusion . . . 61

6 Dynamic Parameters **63**
6.1 Dynamic S value . . . 63

6.2 Experiment 66
6.3 Results 67
6.3.1 Experiment 1 68
6.3.2 Experiment 2 69
6.3.3 Experiment 3 70
6.3.4 Experiment 4 71
6.4 Conclusion 72

7 Conclusion and Future Work **73**
7.1 Conclusion 73
7.2 Future work 74
7.2.1 Synergy with Fast Tsetlin Machine 74
7.2.2 Python bindings 75
7.2.3 Implement Regression 75
7.2.4 Look at improved search mechanics for s parameter 75
7.2.5 Complete CUDA implementation 75

References **81**

Appendices **83**
A Hardware Specification 83

Chapter 1

Introduction

1.1 Motivation

The use of machine learning is growing by the days [40]. Never before have there been so much; data available; powerful computers; demand for tools and competence. In the middle of all of this, some key techniques have gained popularity, namely neural networks [50]. One of the main issues with Neural Networks is that they are hard to debug [48]. As a result, it is hard to understand why the model chooses a given output [48]. Moreover, after Prof. Ole-Christoffer Granmo released his paper on the Tsetlin Machine [16], some work has been done on the Tsetlin Machine. It has proven capable of handling large datasets and solves many difficult classification problems [16] [5]. Just as important, the Tsetlin Machine is more transparent than Neural Networks. Consequently, allowing data scientists to better understand the reasoning of the output from the algorithm [16].

Nevertheless, neural networks are more widely used throughout the industry than the Tsetlin Machine for classification tasks. One of the key reasons for this is that the industry has tools available to assist developers in making machine learning models, like Tensorflow [52]. In order to move the Tsetlin Machine one step closer to being adopted by the industry, a common architecture is required that allows for rapid development and scalability [7].

The motivation behind this thesis is to look at what parts of the Tsetlin Machine algorithm that can be optimized and improved in order to move the algorithm

beyond the academic community. This thesis focuses on the architecture of executing the Tsetlin Machine algorithm both technically and on paper in order to find common ground. In addition, performance will be a key focus throughout this thesis, since that would be one key factor to allow the Tsetlin Machine to handle more advanced tasks.

1.2 Thesis definition

The work shown throughout this Thesis is tightly connected to the following Goals and Hypotheses. These Goals and Hypotheses are listed in their own following sections.

1.2.1 Thesis Goals

The main goals of the thesis are divided into five sub-goals.

Goal 1: Provide an introductory description of how the Tsetlin Machine algorithm works.

Goal 2: Propose a new architecture for the Tsetlin Machine that allows for parallel execution on CPU and the use of multiple GPUs.

Goal 3: Investigate CPU parallelism versus GPU parallelism on various datasets.

Goal 4: Investigate CPU versus GPU on the "counting votes" phase of the Tsetlin Machine algorithm and when to use which.

Goal 5: Investigate whether dynamic parameters allow the model to find optimal parameters or not.

The first goal is to provide a simplified description of the core mechanics in the Tsetlin Machine algorithm. This could allow a broader audience to understand the inner mechanics faster than reading the current papers [16].

The second goal is to rewrite the core algorithm to allow for parallel execution on the CPU and on multiple GPUs. The effect of this is utilizing the resources

on the system that is executing the algorithm. In addition, it would allow the algorithm to handle even more complex problems.

The third goal is to utilize the results from Goal 2, in order to see if there are any performance differences related to executing the Tsetlin Machine on various datasets.

The fourth goal is to analyze a specific phase in the Tsetlin Machine algorithm, called "counting votes". This phase is interesting in particular since it is a vital part of the algorithm.

The last and fifth goal is to take a closer look at whether dynamic parameters help the model to find the best parameters or not. Effectively, allowing the algorithm to adjust its own parameters in order to gain a higher learning rate between each epoch.

1.2.2 Hypotheses

In addition to the thesis goals, the thesis will test the following Hypotheses.

Hypothesis 1: The GPU reduction helps execution speed on larger datasets than CPU reduction.

Hypothesis 2: Dynamic parameters allows the algorithm to learn a good parameter value during training.

Hypothesis 1 states that GPU reduction helps execution speed on larger datasets compared to CPU implementations. Thus, making GPU a preferred choice if the datasets become large.

Hypothesis 2 states that during training, the algorithm should be able to learn a good parameter. In other words, the algorithm should be able to find a good parameter for the model during training and then use it for training.

1.2.3 Scope & Limitations

During this thesis, more advanced iterations of the Tsetlin Machine has been released. These include The Fast Tsetlin Machine [19]. However, this thesis

will focus on the implementation that was first introduced in the original Tsetlin Machine paper [16].

The datasets that were investigated were datasets that have already been compared to other versions of the Tsetlin Machine. Therefore, data-preprocessing is not a part of this thesis.

1.3 Research Methodology

The purpose of this thesis is to investigate what needs to be done in order to improve the Tsetlin Machine algorithm. Several aspects of the algorithms were investigated and almost all works related to implementing the Tsetlin Machine were re-written from the ground in order to fit the desired goals of this thesis.

Each of the chapters has its own contribution to this thesis. The experiments performed in each of the chapters were considered independent from one another. Meaning, they do not depend on one another. Each chapter has a fixed structure that adheres to the following structure:

1. **Introduction** to the problem. The problem is introduced and justified in order for the reader to understand why this is relevant to the Tsetlin Machine algorithm.

2. A **Proposed Solution** is presented in order to solve the problem.

3. The **Experiment** needed to prove this solution is described.

4. Following are the **Results** from the experiments presented.

5. Lastly, a **Conclusion** is given to the problem and its proposed solution, based on the results from the experiment.

The processes and proposed solutions in this thesis mainly uses Unified Modelling Language (UML) to illustrate processes [49]. These graphs have been created using a tool called PlantUML [47].

1.4 Contributions

In the machine learning field, there exists many methods and algorithms to solve various problems. One of the most popular approaches is the use of Artificial Neural Networks [13]. These techniques have several large frameworks and tools that enable the developer to quickly utilize very complex algorithms in an easy and reliable way. Tensorflow [52] and Keras [11] are examples of such tools. By having these tools, the industry can quickly implement Neural Networks into its daily operations at big scales.

In 2018, Prof. Ole-Christoffer introduced the Tsetlin Machine algorithm [16]. From his research, new techniques and algorithms were created. Some of them being capable of solving many of the same tasks as Neural Networks were capable of. However, tools for solving these tasks using the Tsetlin Machine algorithm were still lacking.

This thesis proposes some changes to the Tsetlin Machine algorithm. Thus, bringing it closer to generalize the concept and making tools for the algorithm. In addition, providing an easy architecture allows for more research and development to be done on the algorithm.

With the Tsetlin Machine allowing to change its own hyper-parameters, it becomes much simpler to create new models and approach new problems knowing that most of the advanced mechanics will adjust itself to fit the problem.

1.5 Thesis outline

The thesis is divided into several chapters where the main topic will be as follows:

- **Chapter 2** provides some essential background information that is relevant to some of the main topics throughout this thesis.
- **Chapter 3** investigates the current state-of-art surrounding the Tsetlin Machine and provides an introduction to the algorithm.
- **Chapter 4** explains the proposed architecture and how parallelism would affect the algorithm.

- **Chapter 5** proposes a new method of counting votes within the Tsetlin Machine using parallel reduction.
- **Chapter 6** shows how dynamic parameters can be utilized to allow the algorithm to learn its own parameters.
- **Chapter 7** concludes this thesis and provides some insights on possible future work.

Chapter 2

Background

In the field of machine learning, there exists many algorithms and techniques for solving the same problem. One of the most common lately is called neural networks. However, neural networks can be very complex to understand and hard to compute. As a result of that, a new trend has started that desires the algorithms to explain in deeper details why a model has concluded with a given action or result [23] [48]. This is the case especially in medical appliances, where for example a doctor needs to know why a patient has been classified as having cancer [39].

Another important aspect of machine learning is the required amount of training data that is required for the model to learn a generic approach to a problem. As problems grow, the model requires more and more data. These problems remain difficult to solve, mostly due to the lack of data available [23].

2.1 Neural Networks

Neural Networks are a set of nodes (neurons) that are connected to each other in a dense network. We call this network an Artificial Neural Network (ANN) [32].

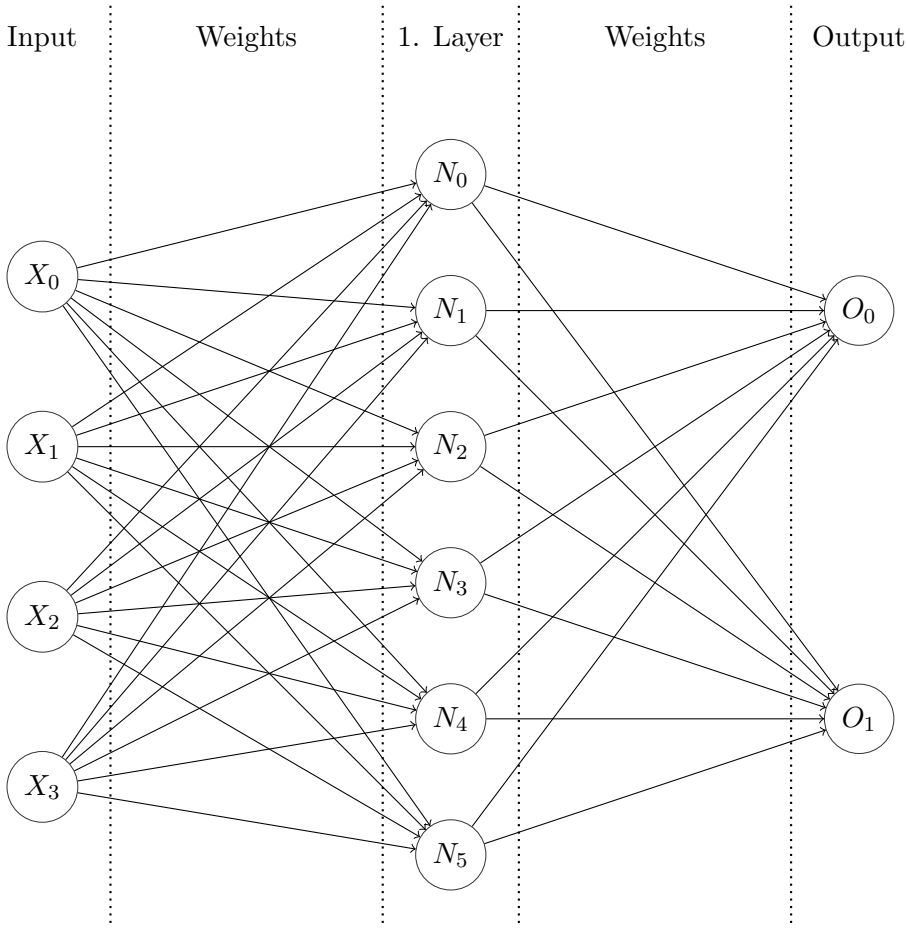

Figure 2.1: A single layer neural network.

In Figure 2.1, the inputs are the different features of a given sample that are passed to the model during training. The lines between the nodes in Figure 2.1 are called weights. The weights are just some mutable decimal numbers that are multiplied with the output from the previous node, resulting in a new value that will be processed by the node and its sum function. The output from this node will be calculated from the activation function, resulting in one of the two outputs being activated [33].

During the training, the weights (which are mutable) change. The weights change in order to adjust the correct output to be triggered based on the given input values [29]. In the end, we are left with some weights that fits the model [42].

However, in order to calculate the "correct" weights for a model, we need to perform a big amount of computation [53] [41].

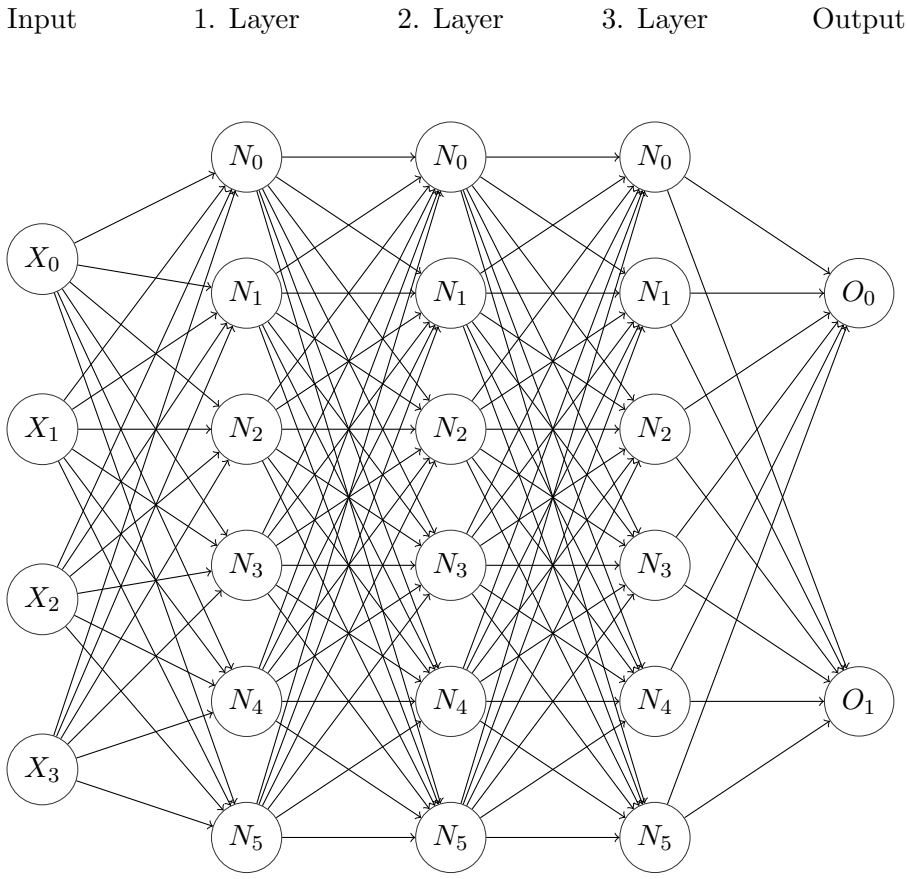

Figure 2.2: A small deep neural network (DNN).

Neural Networks is not limited to "one single layer" of nodes. In fact, they may contain many more. This is called Deep Neural Networks and the technique is called Deep Learning [37]. In Figure 2.2, there are many more layers that work together to solve a problem. Gradually, like many other algorithms, when the model grows, the required computational work increases [12].

As a consequence, when multiple layers are introduced, like in Figure 2.2, the model becomes less transparent and it is harder to understand why the output was selected by the model. As a result of this, new models are being proposed

that try to increase transparency and to reason more like humans [30].

2.2 Learning Automaton

The Learning Automaton is in short, a simple decision-making mechanism that can learn from previous experiences and adapt to those [17]. These automatons consist of a mutable state that may yield an output. Understanding how a Tsetlin Automaton works is fundamental to understanding the Tsetlin Machine and its concepts.

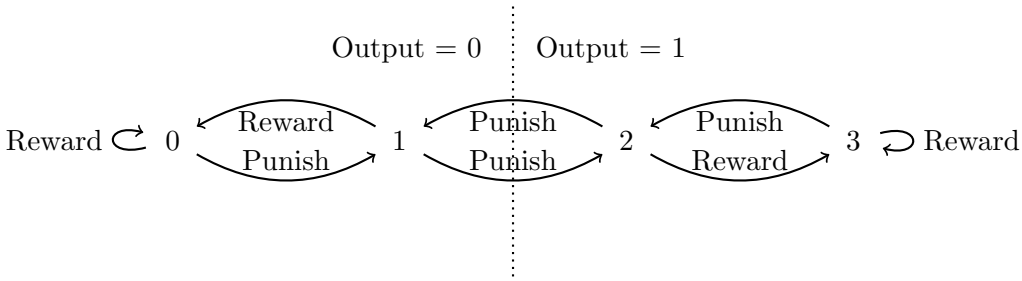

Figure 2.3: One Tsetlin Automaton and how the states describes what action/output it will yield.

As shown in Figure 2.3, an automaton may have several states (in this case four), where the state indicates what the output from the Automaton will become if asked. In order to compute what the output will become when asked, we need to know the current state and the max state.

2.2.1 Input data

The Learning Automaton is a very simple unit. Therefore, it only accepts a single discrete value of either true or false ($\{1, 0\}$), both during training and when evaluating the Automation. This makes it ideal for the computer since it allows the computer to utilize a single bit to represent the input data for the Automaton. Although, this often involves more prepossessing of both the training data and evaluation data that is associated with the Automaton.

2.2.2 Training process

During training, the Automaton starts with a random value between 1 and the $MaxState$. $MaxState$ is a parameter that is set during the creation of the Automaton. In fact, the Automaton will only receive *reward* or *punishment* events. These events are determined by the feedback mechanism. Usually, a bad output from the Automaton should result in a *punishment*, while a correct action should result in *reward*. Furthermore, it is important to state that this is not always the case as.

$$\text{Action}(A_s, F) = \begin{cases} A_{s+1}, & \textbf{if } 1 \leq s \leq \frac{2}{\text{MaxState}} \textbf{ and } F = \text{Penalty} \\ A_{s-1}, & \textbf{if } \frac{2}{\text{MaxState}} < s \leq \text{MaxState} \textbf{ and } F = \text{Penalty} \\ A_{s-1}, & \textbf{if } 1 < s \leq \frac{2}{\text{MaxState}} \textbf{ and } F = \text{Reward} \\ A_{s+1}, & \textbf{if } \frac{2}{\text{MaxState}} < s < \text{MaxState} \textbf{ and } F = \text{Reward} \\ A_s, & \textbf{otherwise}. \end{cases} \tag{2.1}$$

Feedback can be given to the Automaton by an external referee or evaluation function. Based on the feedback, the state of the Automaton may either decrease, increase or stay the same position. Like shown in Equation 2.1, the Automaton state is given as A_s and the feedback is given as F. Either the feedback will be *reward*. Otherwise, it would be *punishment*.

2.2.3 Output

$$Output(A_s) = \begin{cases} 0 & \textbf{if } 1 \leq s \leq \frac{2}{\text{MaxState}} \\ 1 & \textbf{if } \frac{2}{\text{MaxState}} < s \leq \text{MaxState}. \end{cases} \tag{2.2}$$

To evaluate the automaton, we use an output function, see Equation 2.2. If the function evaluates to 1, we know that the automaton's output is 1. Equally, if the function evaluates to 0, the output is 0.

In order to facilitate learning, the Learning Automaton are able to learn by its action using either *punishment* or *reward*. Recall Figure 2.3, here we can observe that any reward feedback will change the state in a direction further away from the middle. However, any punishment feedback will move the state further into the center, and eventually on the other side.

2.3 Collective of Learning Automatas

In Section 2.2, the dynamics of one single Automaton is described. In this section, the mechanics of many Automata are described in detail.

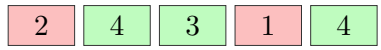

Figure 2.4: A small collective of 5 Automata, where the max state is 4, ranging from 1 to 4 inclusive. The output of each Automata is highlighted in different colors; green represents 1; red represents 0.

.

As seen in Figure 2.4, there are two Automata that have an output of 0, while the other three have an output of 1. There are several ways the collective can have a consensus on an output. Some of them are as follows.

- **Majority vote** where the majority of the outputs will be the final vote. In Figure 2.4, a majority vote would result in the output of 1. This approach is often used for discrete classification where the output needs to be absolute.
- **Counting votes** could be utilized when the model wants to find out how much something matches a model or does not match a model. From Figure 2.4, the result could have been either 3 if it was interesting in finding similarities or 2 if it was interesting in finding differences.

One common application or experiment is called The Gur Game [54]. In this game, the Automata acts as a decentralized system that each controls whether a sensor should be turned on or off. The sensors are controlled by the Automata output, where an output of 0 indicates the sensors to turn off, while an output of 1 indicates the sensors to be turned on. The goal is for all the sensors to conserve power, and not have any more than the desired amount of sensors turned on. However, a minimum number of sensors needs to be turned on in order to get a good reading. In addition, there is no communication between the Automata since this is a decentralized system.

The solution for this was presented by M.L Tsetlin [54]. Thus, proving the power of multiple Automata working together, even when they have no knowledge of one another.

2.4 Logical gates

The use of logic gates is some of the most fundamental parts of a computer. Ranging from various gates that can perform different calculations, there are two interesting gates regarding this thesis, namely the *AND*, and the *IMPLY* gate [31] [36]. Sometimes these gates are called operations based on the use case.

$$\text{AND}(a,b) = \begin{cases} 1 & \text{if} \quad a = 1 \wedge b = 1 \\ 0 & \text{if} \quad a = 0 \vee b = 0 \end{cases} \tag{2.3}$$

a	b	$a \wedge b$
0	0	0
1	0	0
0	1	0
1	1	1

Table 2.1: Truth table for AND gate/operation

$$\text{IMPLY}(a,b) = \begin{cases} 0 & \text{if} \qquad a = 1 \wedge b = 0 \\ 1 & \text{otherwise} \end{cases} \tag{2.4}$$

a	b	$a \implies b$
0	0	1
1	0	0
0	1	1
1	1	1

Table 2.2: Truth table for IMPLY gate/operation

Understanding how both the AND (Equation 2.3), and IMPLY (2.4) works is important for understanding the Tsetlin Machine. Truth tables for both operations are included as Table 2.1 and Table 2.2.

2.5 Kernels

Kernels are a fundamental part of GPU programming. Thus, understanding how each kernel executes is crucial to understand the choices made during this thesis. Therefore, this section will briefly explain the mechanics of a kernel.

A kernel is simply just a function that is being executed on multiple processors in a GPU. This function does not look like a traditional function from regular programming. For instance, it cannot return a value. In order to store results, the logic for persisting the results from computations needs to be assigned to each processor [24] [9].

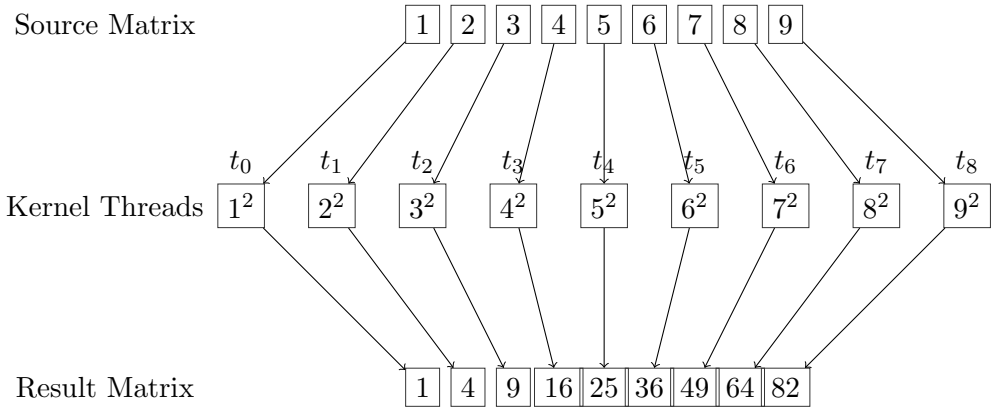

Figure 2.5: A simple GPU Kernel that squares all values in a matrix and stores them in a new matrix.

In Figure 2.5, the source matrix of the values 1..9, is to be squared. In traditional computing, this is easily achieved using a "for loop". However, in GPU programming, each of the elements can be calculated at once on each of their own processors. The results from the computation are stored in a new matrix, that is accessible from the program.

This is just a trivial example that might not yield any improvements at all, but imagine matrices with millions of elements running being calculated on a single processor versus being calculated on a GPU in a kernel. This is where the GPU outperforms CPU by a large magnitude.

Chapter 3

Introduction to the Tsetlin Machine

This chapter provides an introductory description of the Tsetlin Machine algorithm and its mechanics [16]. Moreover, some new abstract techniques for both performing evaluation and training are introduced to simplify even more. This chapter is related to Goal 1 of the thesis.

3.1 Multi-class Tsetlin Machine

In April 2018, Prof. Ole-Christoffer Granmo published the first paper regarding The Tsetlin Machine [16]. This paper explains the mechanics of Tsetlin Automatons and how they can be utilized in larger models and therefore being able to tackle more complex tasks than before. However, it did not take long before several revisions were submitted. One of them included the Multi-class Tsetlin Machine that this thesis focuses on [16] [5].

The mechanics of The Classical Tsetlin Machine builds upon the topics introduced in Chapter 2. Using those mechanics, The Classical Tsetlin Machine is a pure classification algorithm that tries to generate conjunctive clauses that can be utilized in order to check if a sample belongs to a class or not. With this base, the Classical Multi-class Tsetlin Machine was introduced. The main difference being able to classify between multiple classes/categories and not only two.

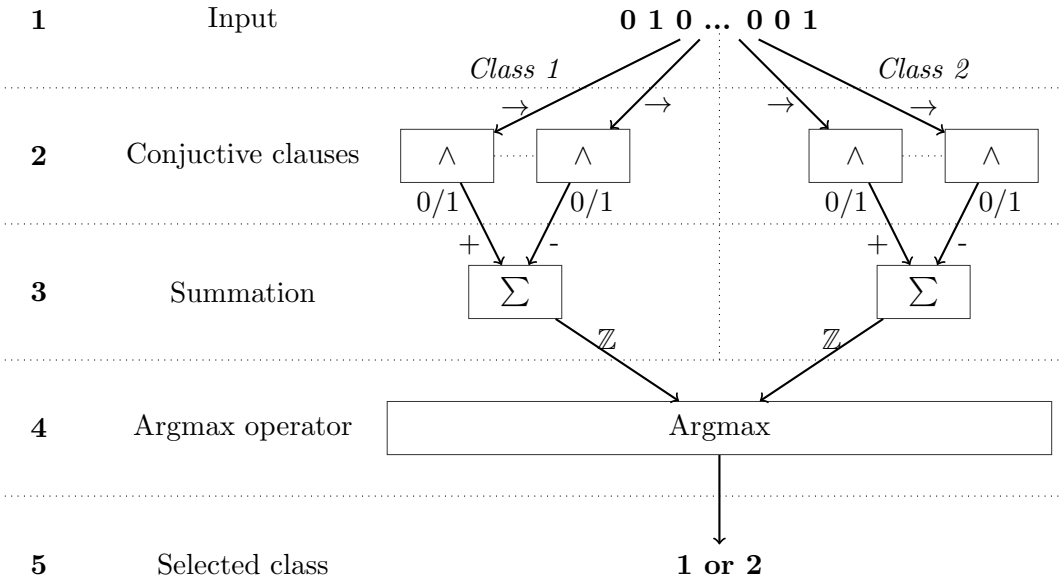

Figure 3.1: The Classical Multi-class Tsetlin Machine.

As seen in Figure 3.1, the procedure is mainly divided into 5 steps. These steps are briefly described in the following way:

1. The input is provided as a 1D array/list consisting of the values 0 or 1. These are the features or characteristics of a sample that is provided to the model.

2. The conjunctive clauses are evaluated against the input. The possible outputs of each clause are either 0 or 1. A value of 0 tells the model that the sample does not match the pattern that particular clause has learned, while 1 indicates a match.

3. During the summation phase, the output from the previous step is added together using a voting system where the odd indexed clauses either "casts a vote" for this being the wrong class (negative vote) or a blank vote. The even indexed clauses "cast a vote" for this being the correct class (positive vote) or a blank vote. The vote will be either negative or positive if the output of the clause was evaluated to 1 (true) and blank if 0 (false).

4. A number from each of the classes are collected by the argmax step. Here, the largest value is selected as "the most likely class".

5. In the end, the final output is represented as a number that can be mapped to something more meaningful depending on the application utilizing the algorithm.

3.2 Structure

To better understand how the Tsetlin Machine algorithm performs, it is important to understand how the data and the model are structured. These can be structured in various ways depending on the implementation of the Tsetlin Machine algorithm. In order to make the explanation of the algorithm simpler, this thesis only focuses the one implementation described in this chapter.

3.2.1 Input data

Before reviewing the model, it is important to see what kind of data the Tsetlin Machine accepts during training and evaluation. The Tsetlin Machine handles discrete input. In short, only values of either 1 or 0 (true or false) are accepted. Meaning, in many cases, some kind if pre-processing of input data are required. Typically, the input data is represented as a 1D array where the length/size of the array is the number of features that are extracted from the data.

	I	**hated**	**you**	**liked**	**they**	**that**	**movie**	**disliked**	**loved**	**the**
S_0	1	0	0	1	0	1	1	0	0	0
S_1	0	0	1	0	0	0	1	1	0	1
S_2	1	1	1	0	0	0	0	0	0	0
S_3	0	0	0	0	1	0	1	0	1	1

Table 3.1: Shows how documents can be converted into vectors that acts as input data for a Tsetlin Machine model.

In Table 3.1, an example of how these features could be extracted is illustrated by using a "bag-of-words" example [14]. The samples (S_x) are assigned to equally sized vectors where each of the elements are aligned. The value in each element can either be 0 or 1, depending on if the word is present in the sample or not. The

same technique can be applied to pictures, however, this process involves more pre-processing.

3.2.2 Model

The Tsetlin Machine model consists of n given classes. Each of those classes has a 2D matrix which has several components that have different meaning and tasks during both training and the evaluation of the model. The two dimensions of this matrix are as follows:

1. **Clause** - There are no limits to how many Clauses a model could or should have. However, the number of Clauses has a big impact on the training process, if configured badly. Too many Clauses would result in longer training time and more resource allocation, whereas too few could result in some patterns not being detected in data.

2. **Automata** - The number of Automata should always be twice the amount of features (components of the input data). I.e. if the input data has 34 features, the number of Automata should be 68.

In addition to the 2D matrix, the model has some additional values that are essential to know:

- The number of *Classes*.
- The number of *Clauses* per Class.
- The number of *Automatas* per Clause.
- The number of *Features* on the input data.
- The number of *States* per Automata.

It is important to note that in the current state-of-the-art and implementations, these values must be known during compile-time. This also applies to the other variables introduced in this section.

3.3 Training

The training process of the Tsetlin Machine is an interactive process. Therefore, several steps need to be performed in a given order. These are as follows:

1. **Conversion** - Convert the model to a state model.

2. **Evaluate** - Evaluate a sample against the state model.

3. **Count votes** - Count/sum the votes from each of the Classes.

4. **Calculate feedback** - Calculate what type of feedback to give each Clause.

5. **Perform feedback** - Perform the feedback on each of the Automata.

In order to simplify the process, the following steps will focus on how one class, in particular, evaluates one single sample and its "part" of the model.

3.3.1 Conversion

In order for the Automaton to function, a step that involves calculating the output of each of the Automata is required. To achieve that, Equation 2.2 is utilized on each of the Automata in the model.

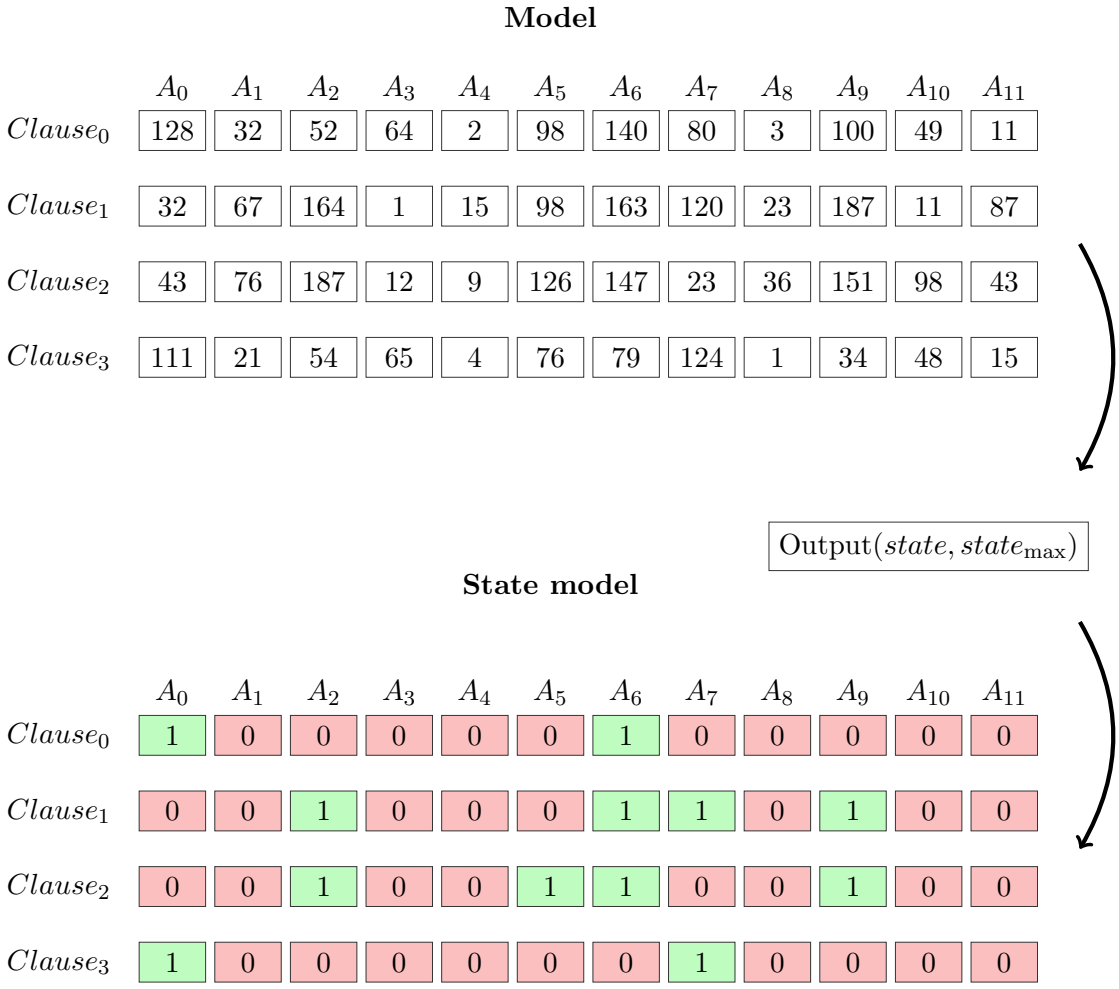

Figure 3.2: The conversion process of a Tsetlin Machine model to a state model. The *Output* function that is utilized is Equation 2.2. In this example, the $state_{\max}$ is set to the value of 200.

Once the action of each of the Automata in the model has been calculated as shown in Figure 3.2, the result is stored as a state model. This state model is used in the next phase when the model should be evaluated.

3.3.2 Evaluate

In this step, the training data is introduced. In the previous step (See 3.3.1), the model was converted to a state model. That state model will be used when evaluating the clause's output values with the sample. However, before that is possible, the sample needs to be "expanded".

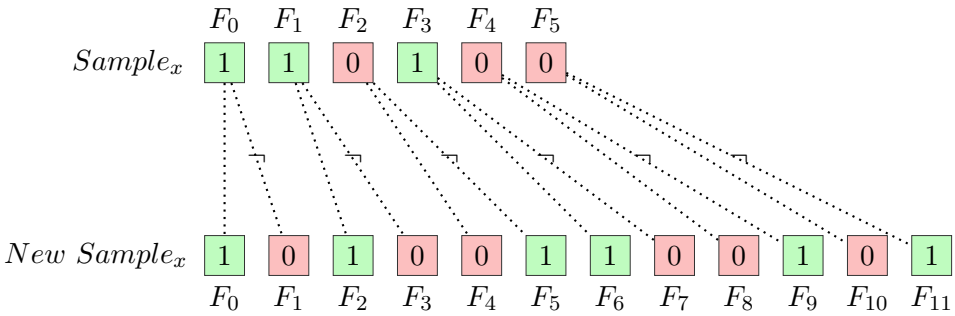

Figure 3.3: A sample with 6 given features, is expanded to a new sample with 12 features.

The proposed sample expansion, takes the value F_x and stores it in the new sample at location F_{2x}, while at the same time negating the value and storing it as well in F_{2x+1}. The sample expansion is illustrated in Figure 3.3, where the total amount of features has grown to twice as many features. This is one important aspect of The Tsetlin Machine that is easily ignored. However, being an abstract concept, the mathematical process is best described using Figure 3.3. Nevertheless, the actual process of expanding the sample vector is a computationally costly process that is not feasible.

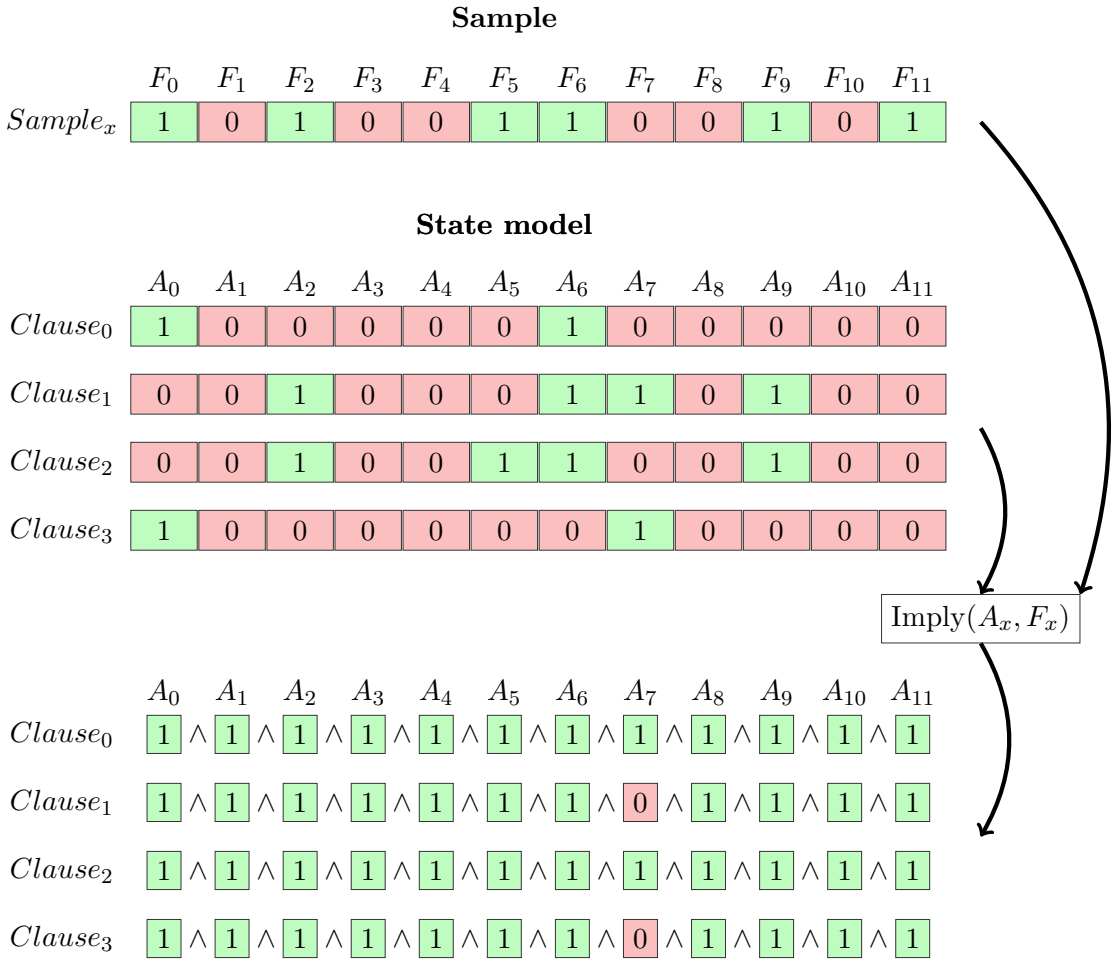

Figure 3.4: A sample and a state model that uses the IMPLY operation on each of the elements to create conjunctive clauses consisting of logical AND operations.

As seen in Figure 3.4, each of the four clause's Automata is implied with the aligned feature from the given sample. Thus, forming four conjunctive clauses built by the output from the IMPLY function. Each of those elements is sequentially passed through the AND gate, resulting in either the output of 1 or 0.

In Figure 3.4, the final output from each of the clauses is as follows:

- $Clause_0$: 1
- $Clause_1$: 0
- $Clause_2$: 1
- $Clause_3$: 0

3.3.3 Counting votes

In this step, the output of each of the Clauses in a Class is stored into one value called V. In addition, the Tsetlin Machine's reduce operation takes into account the index number of the clause to determine what the clause tries to learn. Clauses with "even" numbered indexes are counted as positive votes, while Clauses with "odd" numbered indexes, are counted as negative votes. Because of this, "even" numbers include 0 ($\mathbb{N}_0$). Thereafter, the T (threshold variable) is introduced. This value is used to "clip" the final votes.

$$\mathrm{V} = \sum_{even}^{n} Clause_x - \sum_{odd}^{n} Clause_x \tag{3.1}$$

$$\mathrm{V} = \sum_{x=0}^{n} Clause_x \cdot (1 - (2 \cdot (x \mod 2)) \tag{3.2}$$

This operation can be achieved using one of two formulas. Equation 3.1 is a simplified version of Equation 3.2. The latter is often used in most implementations due to its ease on computer resources. However, the essence of the operation remains the same.

Finally, the Votes (V) value is passed through a threshold function like shown in Equation 3.3.

$$\mathrm{Threshold}(V) = \begin{cases} T & \text{if} \quad V > T \\ -T & \text{if} \quad V < -T \\ V & \text{otherwise} \end{cases} \tag{3.3}$$

The result from Equation 3.3, will be passed on to the phase of the Tsetlin Machine. The T is a hyperparameter that is given to the model at the start of the training process. T is used to help alleviate the vanishing signal-to-noise ratio problem [5].

3.3.4 Calculate feedback

In this step, the Tsetlin Machine will calculate what type of feedback to provide for each of the Clauses. The various types of feedback are as follows:

- **No feedback** will be given to either of the automatas in the Clause.
- **Type 1 Feedback** is given to combat false negatives and amplify true positives.
- **Type 2 Feedback** is given to combat false positives.

$$\text{Feedback}(V,T) = \begin{cases} 0 & \text{if} \quad r > \frac{T-V}{2 \cdot T} \\ 1 & \text{otherwise} \end{cases} \tag{3.4}$$

All of the Clauses will be tested with Equation 3.4. In Equation 3.4 r is a "pseudo-random" number that exists between 0 and to 1 exclusive ($r \in \mathbb{R} \mid r \in [0..1)$). The output of 0 (false) indicates that no feedback will be given to the tested Clause. However, an output of 1 (true) will progress the Clause to test whether it should receive Type 1 or Type 2 feedback.

$$\begin{aligned} \text{even} &= \{x/2 \in \mathbb{N}_0 \mid x <= n \} \\ \text{odd} &= \{x/2 \notin \mathbb{N}_0 \mid x <= n \} \end{aligned} \tag{3.5}$$

$$\text{FeedbackType}(Index, Target) = \begin{cases} 1 & \text{if} \quad \text{Target} = 0 \wedge \text{Index} \in \text{odd} \\ 2 & \text{if} \quad \text{Target} = 0 \wedge \text{Index} \in \text{even} \\ 1 & \text{if} \quad \text{Target} = 1 \wedge \text{Index} \in \text{even} \\ 2 & \text{if} \quad \text{Target} = 1 \wedge \text{Index} \in \text{odd} \end{cases} \tag{3.6}$$

The sets of *even* and *odd* is the sets of all even numbers and odd numbers. These sets can be constructed using the set constructors from Equation 3.5. n is the number of Clauses in each class of the model.

Target is simply a number from 0 to 1 ($\{0, 1\}$). 0 indicates that the sample that is being trained on should not evaluate to the current Class and 1 indicates that the model should evaluate to the current Class.

With that knowledge, and by utilizing Equation 3.6, the feedback type can be determined for each of the clauses in the Class. These values are carried over to the next step where the feedback will finally, be performed.

3.3.5 Perform feedback

The final step of the training process is to actually give the feedback to all of the Clauses and Automata that are subject to feedback. The Clauses that were assigned a type of feedback in the previous process are either subject to Type 1 or Type 2 feedback. The actual feedback process is different depending on whether the execution happens on the CPU or the GPU. However, based on the previous steps, two truth tables lay out the probability of how likely it is for each of the Automata to receive a reward, punishment or nothing based on its output and output of the Clause.

Document → Target Clause:		1		0	
Target Literal:		1	0	1	0
Action 1	P(Reward)	$\frac{s-1}{s}$	NA	0	0
	P(Inaction)	$\frac{1}{s}$	NA	$\frac{s-1}{s}$	$\frac{s-1}{s}$
	P(Punish)	0	NA	$\frac{1}{s}$	$\frac{1}{s}$
Action 0	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(Punish)	$\frac{s-1}{s}$	0	0	0

Table 3.2: Probability table for Type 1 Feedback. [16]

Document → Target Clause:		1		0	
Target Literal:		1	0	1	0
Action 1	P(Reward)	0	NA	0	0
	P(Inaction)	1.0	NA	1.0	1.0
	P(Punish)	0	NA	0	0
Action 0	P(Reward)	0	0	0	0
	P(Inaction)	1.0	0	1.0	1.0
	P(Punish)	0	1.0	0	0

Table 3.3: Probability table for Type 2 Feedback. [16]

From both Table 3.2 and Table 3.3, the probability for what type of feedback that is provided for each of the Automatas in a Clause is given.

3.4 Conclusion

In this Chapter, the Multi-Class Tsetlin Machine was introduced and a description of its mechanics was provided. The algorithm was simplified. For example, a new mechanism to explain the conjunctive clause evaluation against a given input was presented using logic gates. This method gives a gentle introduction to the Tsetlin Machine algorithm. However, like mentioned, the actual implementation varies from implementation to implementation [21] [22] [20] [18]. Nevertheless, the abstract concept remains the same.

This concludes Goal 1 of the thesis, where the goal was to provide an introductory description of how the algorithm works. This approach does not describe the technical implementation, but rather the overall structure and how the algorithm can be mathematically modeled.

Chapter 4

Proposed Architecture

In order to achieve the goals of this thesis, the Tsetlin Machine algorithm required some major changes. Throughout this chapter, a new architecture for the Tsetlin Machine is proposed. The new architecture is divided into two parts, namely CPU and GPU. The main focus with the proposed architecture is to utilize modern CPUs and GPUs ability to do work in parallel according to thesis Goal 2 and Goal 3 [3] [4].

4.1 Model

The proposed structure for the Tsetlin Machine model consists of one single 3D matrix. Each of the dimensions has, like the current Tsetlin Machine, their "domain" and responsibilities [16].

1. **Class** - The model has as many components in this dimension as there are Classes (final output variances/labels). I.e. in a binary setup where there are two possible labels, the number of Classes would be 2.

2. **Clause** - There are no limits to how many Clauses a model could or should have. However, the number of Clauses has a big impact on the training process if configured badly. Too many Clauses would result in longer training time and more resource allocation, while too little could result in some patterns not being detected in data.

3. **Automata** - The number of Automata should always be twice the amount of features (components of the input data). I.e. if the input data has 34 features, the number of Automata should be 68.

By utilizing a 3D matrix as a model, the Tsetlin Machine becomes much more modular, since there is just one contiguous array of integers that is used instead of multiple objects like the current implementations today have [22] [20].

Another important aspect of storing the entire model in one matrix is the compatibility between several systems. In this thesis, a CPU architecture and a GPU architecture is proposed. Even though the programming and flow of execution are different, the model remains the same on both implementations.

In addition to the matrix, some knowledge of the model is required:

- The number of *Classes*.
- The number of *Clauses* per Class.
- The number of *Automata* per Clause.
- The number of *Features* on the input data.
- The number of *States* per Automata.

In the proposed architecture of the Tsetlin Machine, these values can be configured during runtime. In contrast, the current implementations of the Tsetlin Machine only allow these to be configured during compile-time [21]. As a result, if any of these values were to change at any given time in a system, the system would have to re-compile the entire code or module where the algorithm resides.

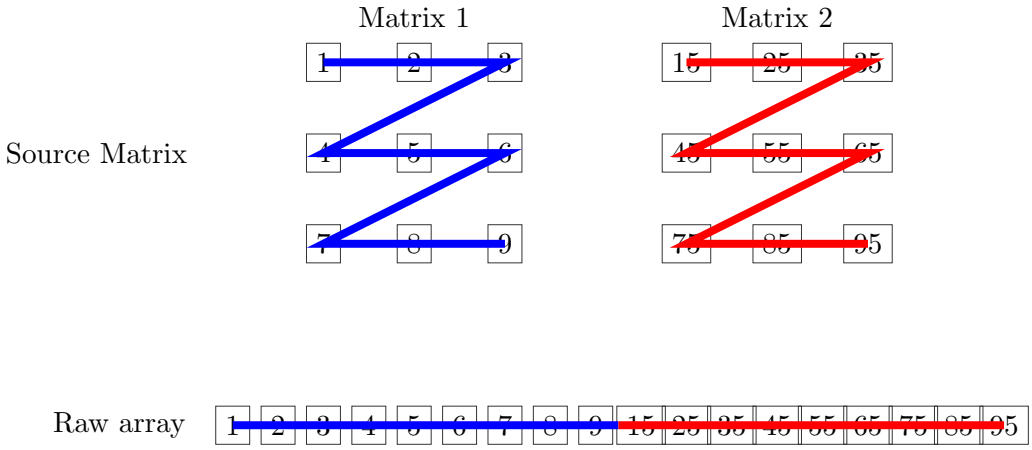

Figure 4.1: 2 2D-matrices represented using one 1D array.

In Figure 4.1, two different matrices are represented using a one-dimensional array. This technique is often referred to as "stride" [10]. By using Figure 4.1's technique of representing multiple matrices in one array, the model gains some advanced features that will be discussed throughout this thesis. In the proposed architecture, this technique is utilized on a 3D matrix. The concept remains the same, only that (from Figure 4.1,) Matrix 1 ($Z = 0$) and Matrix 2 ($Z = 1$), represents the Clauses and Automata from two different Classes, respectively. This technique is widely used in popular libraries like Numpy (Python library) [56].

4.2 CPU

The modern CPU's main advantage is that it has few, but very fast processing units. Often capable of several billion calculations per second. Thus, utilizing a modern CPU's multiple cores are crucial to maximizing the performance and at the same time reducing the amount of time required to train and evaluate [34].

The proposed solution involves using multiple threads under one single process that holds the memory and resources required to complete the training and evaluation [55].

4.2.1 Training

The training process of the Tsetlin Machine is a complex task that involves several steps that are required. First, memory for the training process needs to be allocated. This is only memory that is being used during the training, this includes space for the Clause outputs, votes and what feedback type to give to each Clause.

Thereafter, the training process starts. The amount of epochs to run is given to the model. Multi-threading is utilized in order to speed up training. The amount of threads to create equals the number of classes in the model. Each thread is assigned the task of training their respective class. Inside each thread, a given amount of batches starts. In each batch, all the training samples are being trained on. More details on the training process will be presented later in this section.

Once each thread finishes its training, it is joined back to the main thread which takes the algorithm into the evaluation phase. In this phase, a set of validation data will be evaluated against the model. This allows the model to receive input on how good, accuracy the model has. In addition, precision and recall are calculated for each class. Like before, new threads are created to handle their respective classes. Once created, they loop through the validation samples and saves their votes for each sample in a shared array for each thread. When all the validation samples are validated against, the threads are joined back to the main thread.

The last step involves calculating the accuracy of the model in order to check whether it has improved or not during the training. This is done using the scores from the validation data and checking if the samples with the highest score match the label for that sample.

These steps are repeated for the given amount of epochs. When finished, the allocated memory from the first step. These steps are illustrated in Figure 4.2.

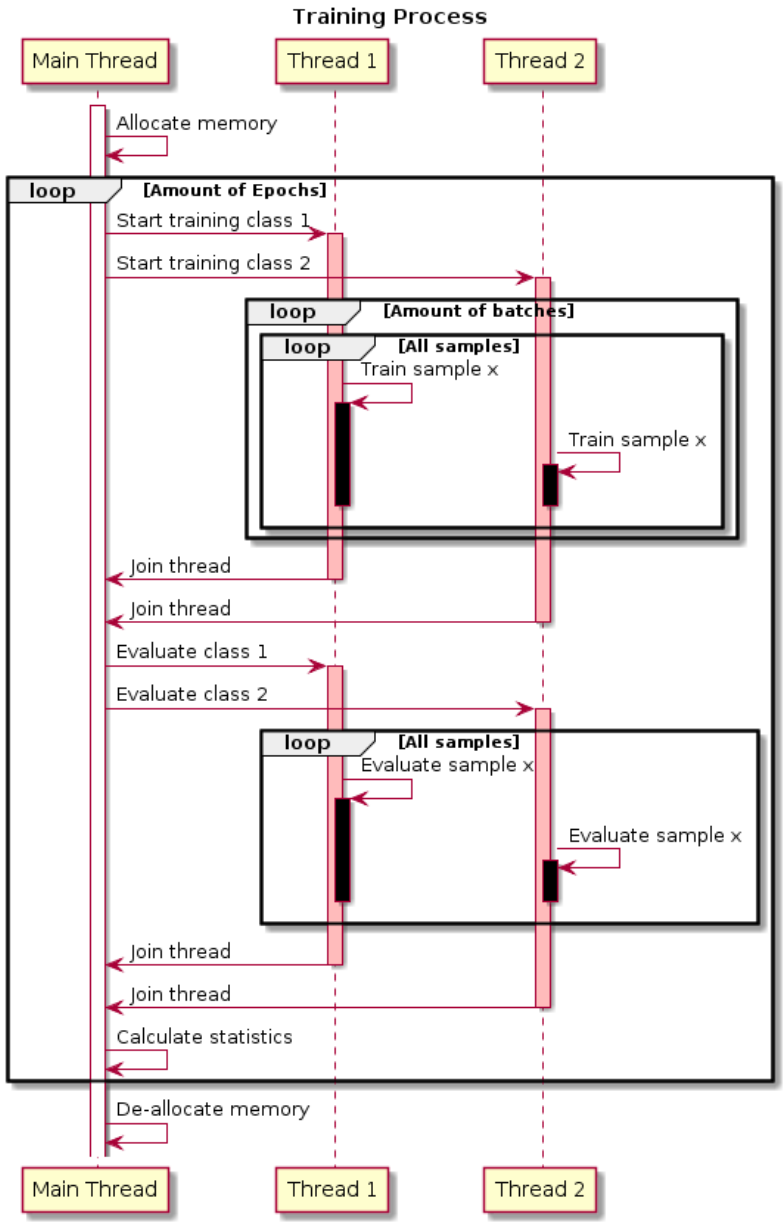

Figure 4.2: The flow of execution with the CPU training process.

As seen in Figure 4.2, the classes are being trained on each of their own threads that belong to the Main Thread. Each of these threads will ensure that its given

class will be trained on all the samples that are given to the model.

Figure 4.2 gives a brief overview of all the steps in the entire training process. However, from here on, the thesis will focus on one of the classes training to give a more detailed view of the process. Diving deeper into the training process of one Class, several steps are performed. During training on one sample, the first step involves validating the Clauses for that Class. Thereafter, the output from Clauses is used to get a score. This score is used when performing feedback. These steps are illustrated in Figure 4.3.

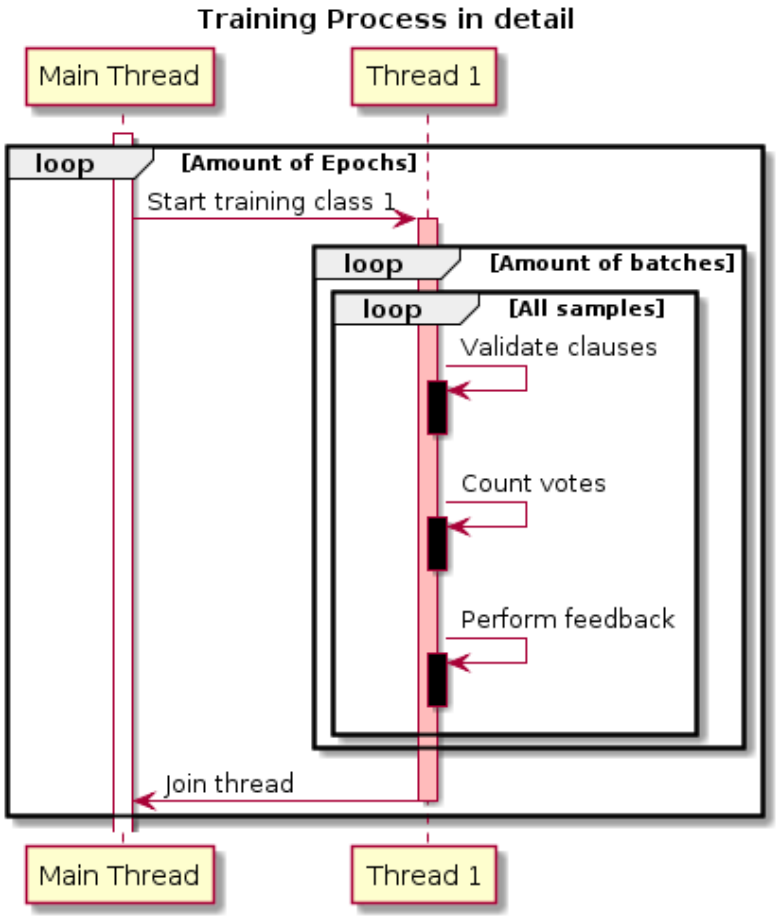

Figure 4.3: The training process in detail for one class on CPU.

The process has been simplified from Figure 3.1 where the algorithm was explained in greater detail. However, in order to simplify the steps and make them more versatile, the steps have been reduced to three parts:

1. **Validate clauses** is validated against the current sample the model is being trained on. The output of each clause is stored for use in the next step.
2. **Count votes** is where the votes are counted for and the result is stored for the next step.
3. **Perform feedback** is the last step for a given sample in the training process. Here all the clauses receive their feedback based on their output in the previous steps.

By simplifying the algorithm into three steps, instead of five, the algorithm becomes simpler to comprehend. Moreover, it makes it easier to focus on each of the steps in terms of what part to optimize. Nonetheless, the underlying steps in the algorithm remain mostly the same, but the general abstraction of the algorithm has been simplified.

4.2.2 Evaluation

During the evaluation phase, threads are created, one for each Class. Thereafter, all the samples that are to be evaluated, are evaluated using the two steps first introduced in Section 4.2.1. Namely, validate clauses and count votes. Since there is no need for feedback when evaluating, this part is not performed during evaluation. This is why evaluating a model is much faster than training on one. These steps are illustrated in Figure 4.4 and remain mostly the same as in the original Tsetlin Machine implementation [16].

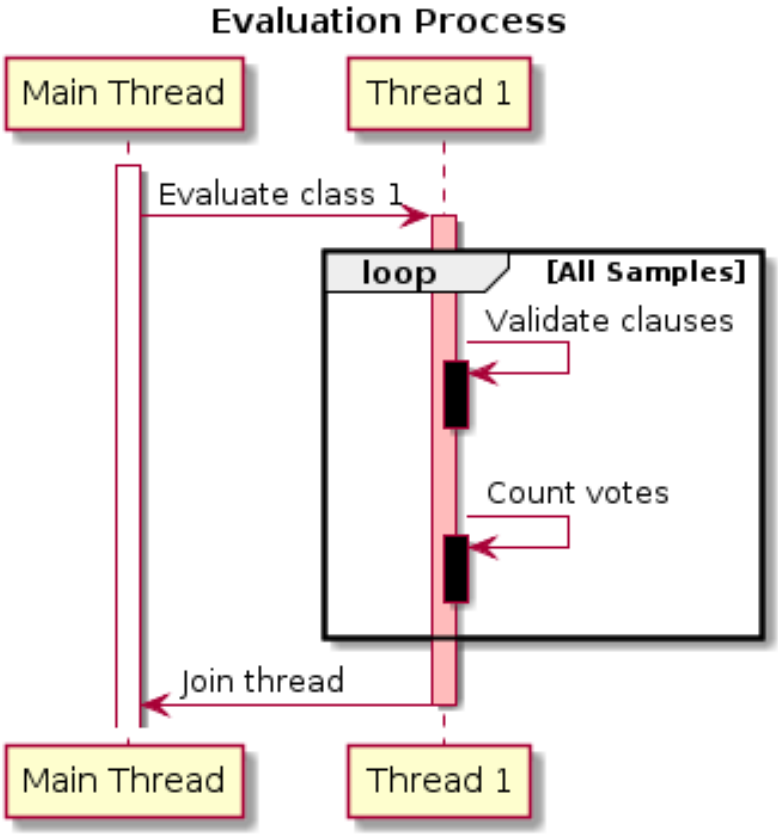

Figure 4.4: The flow of execution with the CPU evaluation process.

4.3 GPU

Utilizing a GPU to its full extent is a very complex and difficult task. There are so many variables to consider, like latency, memory bandwidth, memory banks, and much more [9]. Nevertheless, with proper configuration, the GPU may act as a powerful parallel computing unit [46]. The goals of this thesis do not make that task any easier since variables can be set at runtime. In short, compiler optimizations like definitions is therefore impossible.

In this section, an architectural approach that tries to efficiently utilize one or more GPUs is proposed. There are several problems related to this approach, like how to distribute the data to more GPUs and which GPU is supposed to handle what tasks? In order to solve and explain the proposed approach, this section is divided into several subsections that try to address the issues as they arise with the training.

4.3.1 Distribution of work

Managing how a GPU is utilized in one of the most complex tasks of GPU programming [8]. The underlying system needs some kind of orchestration or idea of how to utilize one or multiple GPUs. One popular way to facilitate the work on GPUs is the use of streams [26] [35]. Streams are both simple and very complex. Therefore, in this thesis, we will be using streams just as a queue of execution. This queue uses a FIFO (First In First Out) algorithm [15]. In short, this means that the first task that is queued is the first one to execute.

With this in mind, it is time to organize the work. Due to the nature of the Tsetlin Machine algorithm, each of the classes has no dependencies of each other during training or evaluation. Consequently, making them an excellent way to separate logic into each stream.

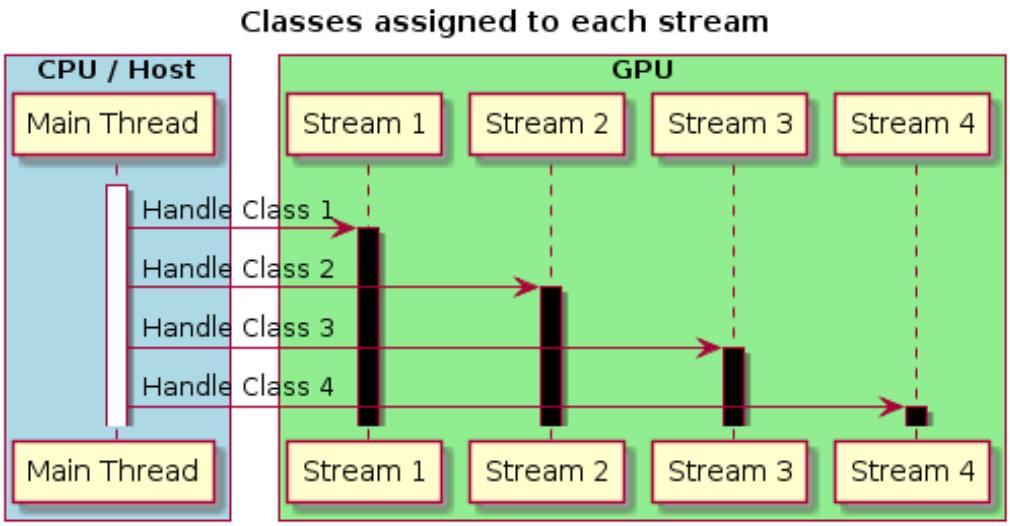

Figure 4.5: Streams tasked with handling one class each on one GPU.

From Figure 4.5 the idea of using one stream per class is illustrated. Streams are pinned to the current GPU upon creation, but there can be more than one stream per GPU. Thus, making the streams act as queues to organize the work for each Class.

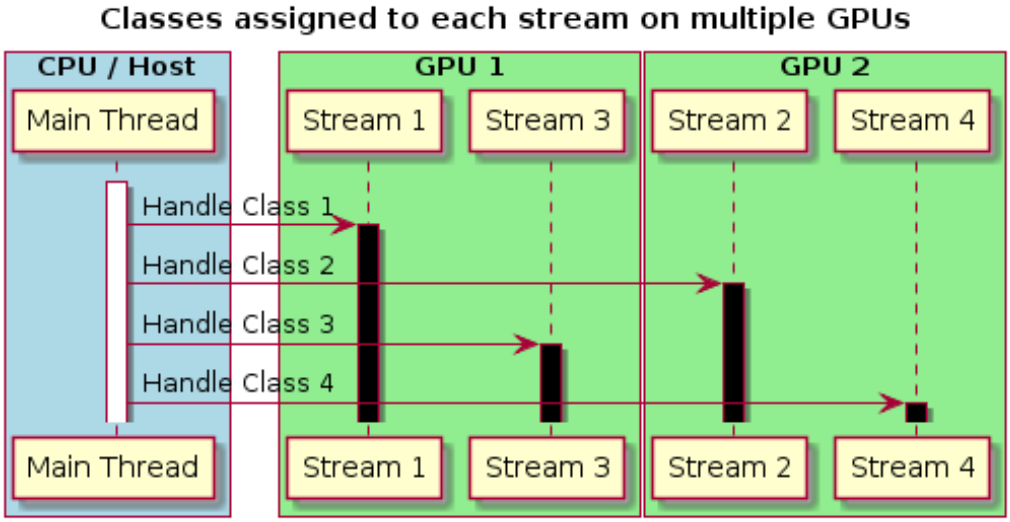

Figure 4.6: Streams tasked with handling one class each on multiple GPU.

Likewise, in Figure 4.6, streams can be distributed onto multiple GPUs.

4.3.2 Kernels

The Kernels in the GPU architecture is designed around the data model. For instance, this means that the kernels should be compatible no matter how many Automata each Clause in the model has. In order to achieve this, a technique called "striding" is utilized [25]. This technique involves that each of the processors in the GPU handles more than one element, or in this case, Automata. However, this becomes relevant when models grow beyond 512 features or the model has more than 1024 Automata due to the warp limits of CUDA [43].

The GPU version has introduced four new kernels that are associated with the architecture. These are as follows:

- **Validate clauses** is responsible for evaluating the clauses against a given sample and storing the outputs in a given array. Thus, making the kernel responsible for handling the second step from Figure 3.1.
- **Count votes** is responsible for reducing the votes into a single score for the

previous kernel. The score is then stored at a given place from the kernel. This makes the kernel responsible for step three from Figure 3.1.

- **Calculate feedback** is responsible for calculating what type of feedback each Clause should receive.

- **Give feedback** handles the feedback that is given to the model during training. This kernel is only used when the model is being trained. Thus, making this kernel responsible for determining the feedback and actually applying it to the model.

With these kernels, the architecture completes the algorithm and is able to perform the required tasks for training and evaluation.

4.3.3 Multiple GPUs

A problem that arises when using multiple GPUs is how to select which GPU for each class. In this thesis, there is no intelligent selecting of the GPUs. However, the distribution of streams to GPUs is used using a simple equation.

$$\text{Device}(C, G) = C \mod |G| \tag{4.1}$$

From Equation 4.1, it is possible to quickly assign a device to the class. Using C as the Class id, and G as a set of available devices (GPUs), it is possible to use the modulo operation on the cardinality of G to assign a device to a class/stream. As a result, the output from Equation 4.1, is the index of the device in G to use (assuming indexes starts at 0).

Class ID	Stream ID	GPU ID
0	2	0
1	3	1
2	4	2
3	5	0
4	6	1
5	7	2
6	8	0
7	9	1
8	10	2
9	11	0
10	12	1

Table 4.1: An example of how the distribution of Streams and GPUs look like on a system with 3 GPUs.

Table 4.1 utilizes Equation 4.1 to assign a GPU to each of the Classes. The Stream ID is just for illustration purposes, they have no coherence with the other values and the actual values would depend on whether the GPUs have other tasks besides the Tsetlin Machine.

4.3.4 Training

Training the Tsetlin Machine model is a complicated process that has been described in Section 3.3. Furthermore, the requirement of running on multiple GPUs makes the problem even more complicated. Figure 4.7 presents an overview of how the training process happens across the entire system with multiple GPUs.

First, memory is allocated to each of the GPUs that are to be used during the training process. Then, the training starts and a loop over all the epochs to train on begins. Like in the CPU implementation, threads for each Class are created on the host. Each of these threads is assigned one GPU to utilize. The first step for these threads is to create/activate one stream that it will use. This allows the stream to act as a queue for the host thread to queue kernel launches.

Once the stream is created, the thread starts to loop the batches and samples like in the CPU implementation, however, now they are being executed on GPUs. At the end of each batch, the host thread waits for each stream to complete its given

tasks. This is called synchronization [26].

Later, when the batches have been completed, the host threads are being joined back to the main thread before the evaluation phase starts. The evaluation phase follows the same patterns as the CPU implementation, one thread is being created for each Class. Again, a stream is created/activated for this class. Following is the evaluation of all the validation samples that have been given to the model. Later, the streams are joined and terminated, and the host threads are joined back the main thread.

Finally, the statistics for the model are being generated based on the evaluation phase. Statistics like, model accuracy and class precision and recall are calculated for each class.

When all the epochs finish, the memory allocated on each GPU is de-allocated. An overall illustration that shows these steps is shown in Figure 4.7.

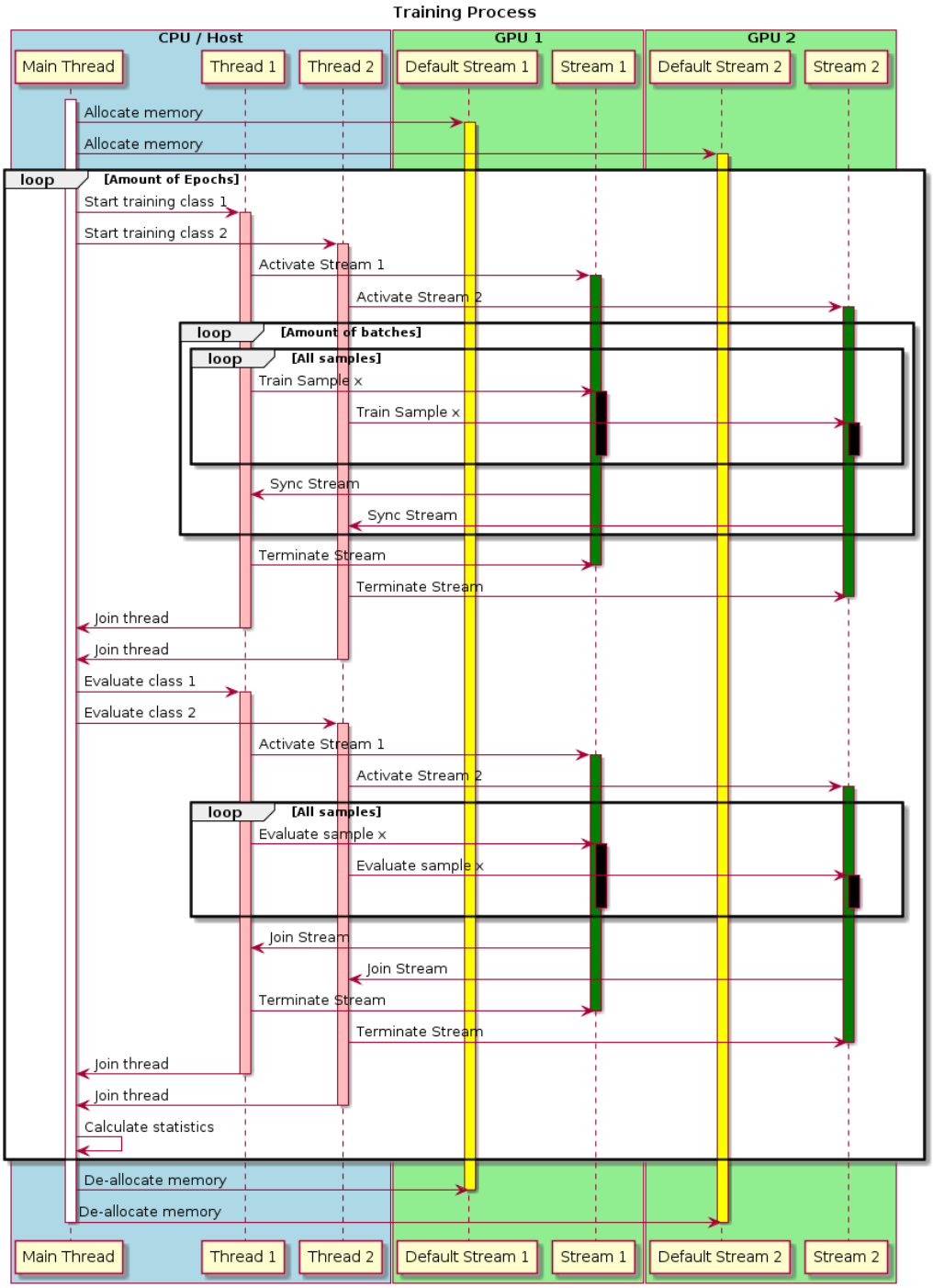

Figure 4.7: The flow of execution with the GPU training process.

As seen in Figure 4.7, the final GPU architecture is more complicated than a pure CPU version. In this example the setup is as follows:

- **The host**, in other words, all code that is executed on the host side (CPU related execution).
- **2 Devices**, in other words, 2 GPUs that are directly attached/connected to the host.
- **2 Classes** or labels that the model should differentiate.
- **4 Streams** in total, 2 that are assigned to the GPUs for memory transfer (default stream), and 2 that are assigned to each of the classes.

Still, there is more happening here that is possible to describe in one figure. Therefore, the following description will focus on one class alone.

In Figure 4.8 the focus is around one Class. This class has one CPU thread, and one GPU Stream. As seen inside the loop of "All samples", there are 4 GPU kernels being launched. These kernels perform the entire training for this class. They are divided into 4 different kernels:

- **Validate clauses** is responsible for calculating the output of all the Clauses when subjecting the given sample to the model.
- **Count votes** is responsible for counting the votes/output from the clauses in the previous step.
- **Calculate feedback** is responsible for calculating the feedback to each Clause.
- **Give feedback** handles the feedback that is to be given to each of the Clauses and Automata.

These steps are repeated for all the samples and batches and eventually the epochs. An illustration of this is shown in Figure 4.8.

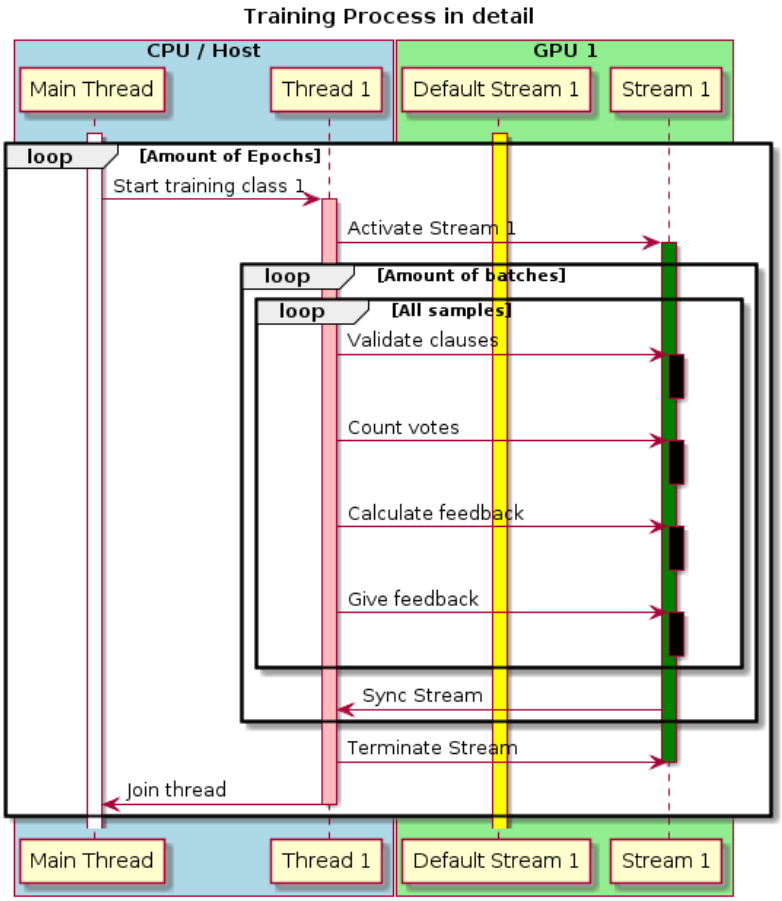

Figure 4.8: CPU and GPU together train one single class in deeper detail.

4.3.5 Evaluation

The evaluation of the samples is illustrated in Figure 4.9. Like in Figure 4.8, Figure 4.9 utilizes kernels that run on the GPU to evaluate a sample. However, when evaluating, there is no feedback involved for the model. Hence, making the process a bit simpler than when training.

The design around the kernels has from the start been about re-usability in order to minimize the complexity of the architecture and its belonging code. As a result, by re-using the kernels from Section 4.3.4, most of the logic can be reused for this

purpose.

Because of this, the two kernels associated with validate clauses and count votes are used for each sample. An illustration of this is provided in Figure 4.9.

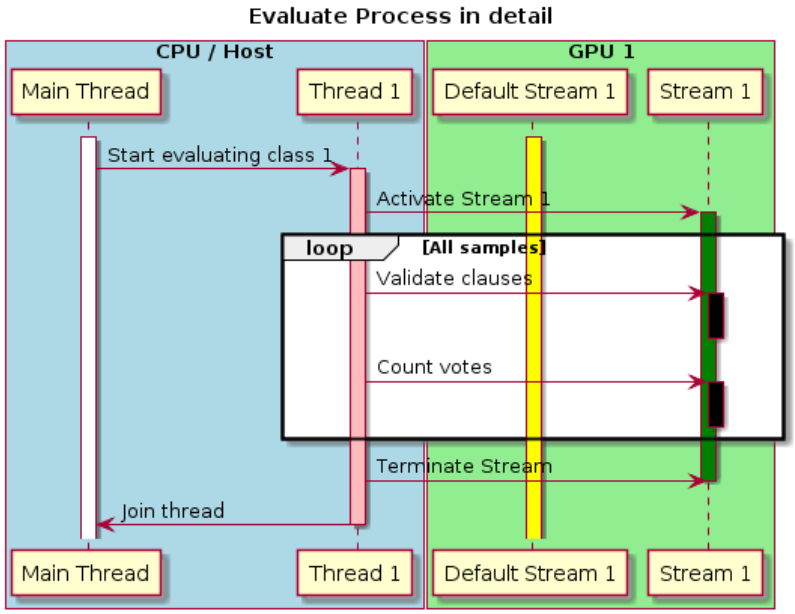

Figure 4.9: CPU and GPU together evaluate one single class in further detail.

4.4 Experiments

In order to test CPU and GPU implementation against one another, two datasets were utilized:

- **Noisy XOR** dataset [22].
- **Iris** dataset [21].

These datasets have been used on other implementations of the Tsetlin Machine [22] [21]. As a result, the datasets have been pre-processed by the repository owners and may, therefore, be biased. However, the validity of the dataset is not the focus of these experiments.

Each of these datasets will be individually tested in their own experiments. First, in experiment 1, the Noisy XOR dataset will be tested. Secondly, the Iris dataset will be tested in experiment 2.

4.4.1 Datasets

The datasets are small in size, but should still allow for the difference to be shown in execution time. The configuration of each of the datasets is listed in Table 4.2.

	Features	**Labels**	**Train. Samples**	**Val. Samples**
Noisy XOR	12	2	5 000	5 000
Iris	16	3	120	30

Table 4.2: The configuration of the datasets used in the experiments for the proposed architecture.

4.4.2 Models

The configuration for the models that will be trained on the datasets is listed in Table 4.3. These configurations are the same as proposed by the repositories where they were retrieved from [22] [21].

	Classes	Clauses	Automata	States
Noisy XOR	2	10	24	100
Iris	3	300	32	100

Table 4.3: The configuration of the models used in the experiments for the proposed architecture.

4.4.3 Training parameters

During training, the parameters used for each dataset and model, are listed in Table 4.4. These parameters are the same as proposed by the repositories where the datasets were retrieved from [22] [21].

	Batches	Threshold	S
Noisy XOR	200	25	3.9
Iris	100	10	3.0

Table 4.4: The training parameters used in the experiments for the proposed architecture.

4.4.4 Metrics and parameters

In order to validate Goal 3, several tests will be performed on the two datasets. The metric to be measured is the execution time per epoch. This metric enhances the view on how the CPU performs versus single and multiple GPUs.

On experiment 1, the following tests were performed:

- Multi-threaded **CPU**, one thread per class.
- **1 GPU** shared among both classes.
- **2 GPUs**, one per class.

On experiment 2, the following tests were performed:

- Multi-threaded **CPU**, one thread per class.

- **1 GPU** shared among all three classes.
- **2 GPUs**, shared among all three classes.
- **3 GPUs**, one per class.

All the mentioned tests will be performed over a period of 100 epochs to make up for noise that could be introduced on the system which could affect runtime.

4.5 Results

In this section, the results from experiment 1 and experiment 2 are presented. The results have been divided into each of their own subsections [1].

4.5.1 Experiment 1

	CPU	**1 GPU**	**2 GPU**
Epochs	100	100	100
Mean	2.344687	29.464554	12.364357
Std	0.325126	1.130496	0.256826
Min	2.023810	26.425700	11.818900
Max	3.163360	32.455300	13.153100

Table 4.5: The results when measuring the execution time from experiment 1 with a multi-threaded CPU implementation, 1 GPU and 2 GPUs.

From the results in Table 4.5, the multi-threaded CPU version outperforms the GPU implementations. However, by adding more GPUs to the execution, the total runtime is reduced. A plot of the runtime across epochs along with average helplines is provided in Figure 4.10.

[1] More detailed results and a Jupyter Notebook are available at this thesis's Github repository [44]

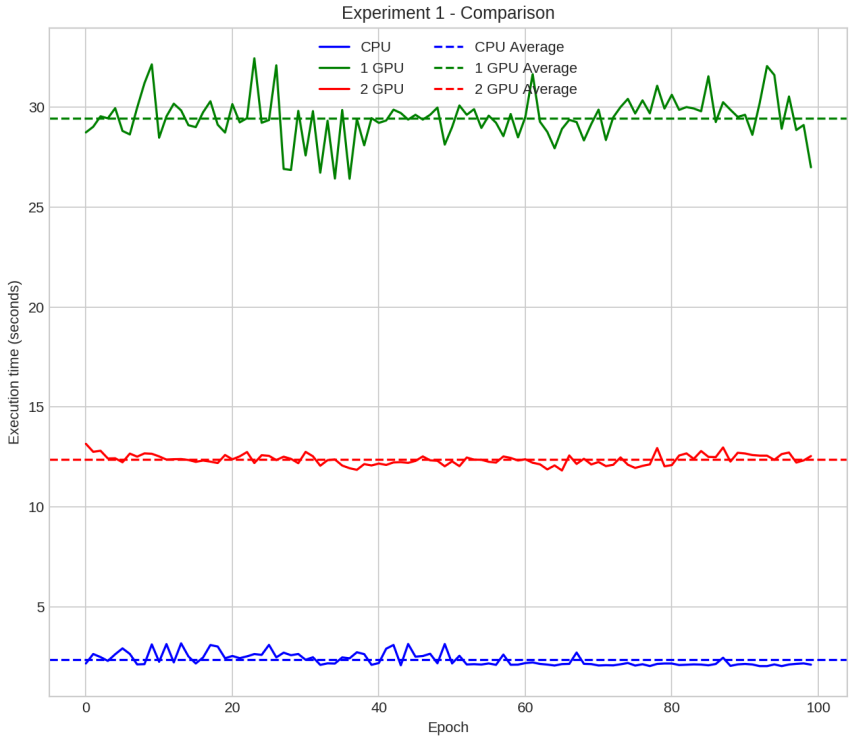

Figure 4.10: The execution time of an entire epoch for each of the configurations in experiment 1.

4.5.2 Experiment 2

From the results in Table 4.6, the multi-threaded CPU version outperforms the GPU implementations. However, by adding more GPUs to the execution, the total runtime is reduced. A plot of the runtime across epochs along with average helplines is provided in Figure 4.11.

	CPU	1 GPU	2 GPUs	3 GPUs
Epochs	100	100	100	100
Mean	0.716166	4.712603	3.279649	2.439178
Std	0.077709	0.484669	0.527712	0.665278
Min	0.600583	3.343220	2.259240	1.564340
Max	0.931968	5.619580	5.007240	3.648040

Table 4.6: The results when measuring the execution time from experiment 2 with a multi-threaded CPU implementation, 1 GPU, 2 GPUs, and 3 GPUs.

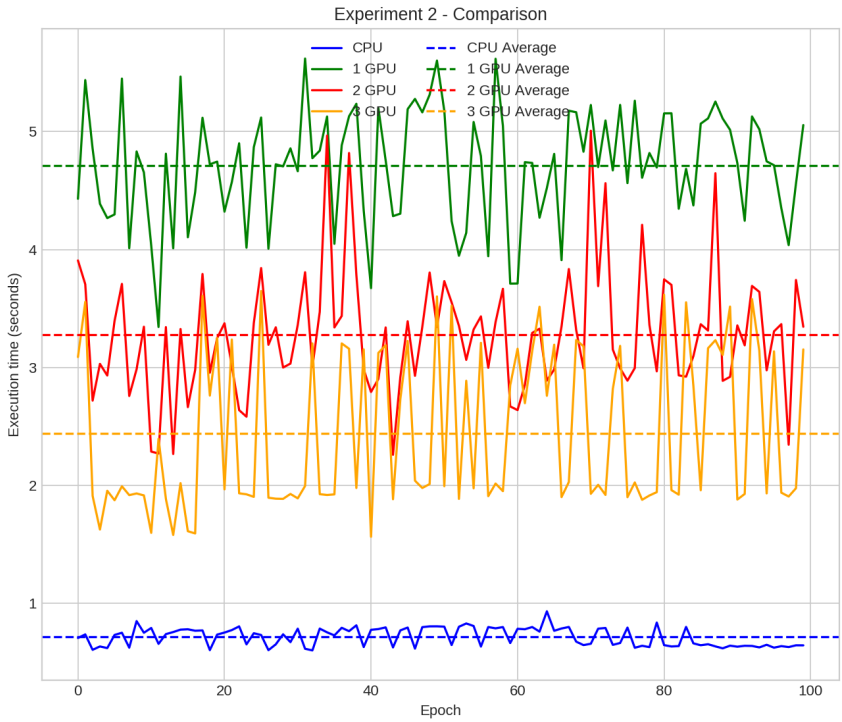

Figure 4.11: The execution time of an entire epoch for each of the configurations in experiment 2.

4.6 Conclusion

Surprisingly, the multi-threaded CPU implementation outperformed both a single and multiple GPUs in both experiments. This could be because of the limited dataset sizes, thus making massive parallelism slower due to the limited amount of work available for each GPU thread.

However, as expected, multiple GPUs performed better than a single GPU. This was clearly shown in Figure 4.11, where the average execution time was cut by more than a second going from 1 to 2 GPUs. The performance gain from 2 to 3 GPUs was not that significant.

This concludes Goal 2 and 3 of the thesis. Goal 2 was to propose a new architecture that allowed for parallel execution of the Tsetlin Machine. This was performed in Section 4.4, and since this yielded results, the goal should be considered complete. Goal 3 was to investigate CPU parallelism versus GPU parallelism. This was performed in Section 4.4 and the results from this comparison was shown in Section 4.5.

Chapter 5

CPU and GPU reduction

In the Tsetlin Machine, the reduction is used to count the votes during the second step in the training process [16]. While the current implementation is in small cases great, problems arise when the numbers of clauses increase beyond thousands and into millions [51]. The solution for this is to utilize parallel reduction in the counting phase of the Tsetlin Machine. This chapter will take a closer look at Goal 4.

In this chapter, a solution for how to utilize the GPU to perform a parallel reduction of votes is introduced. In addition, a performance comparison of the proposed solution and traditional looping with various sized arrays was performed.

5.1 The problem with regular counting

In the Tsetlin Machine, votes are an important factor that is computed many times during the training process (this is described in section 3.3). To perform this action, Equation 3.1 or Equation 3.2 is utilized.

The biggest issue with the current architecture of the Tsetlin Machine is that the counting is processed sequentially by one processing unit [16]. Thus, resulting in longer execution time as the amount of data grows.

Figure 5.1 illustrates how this becomes a problem when the number of Clauses grows into the thousands and beyond. Each of the outputs is sequentially calcu-

lated by one processing unit. Thereby, resulting in a longer execution time that grows by the number of elements to process, as shown in Figure 5.1.

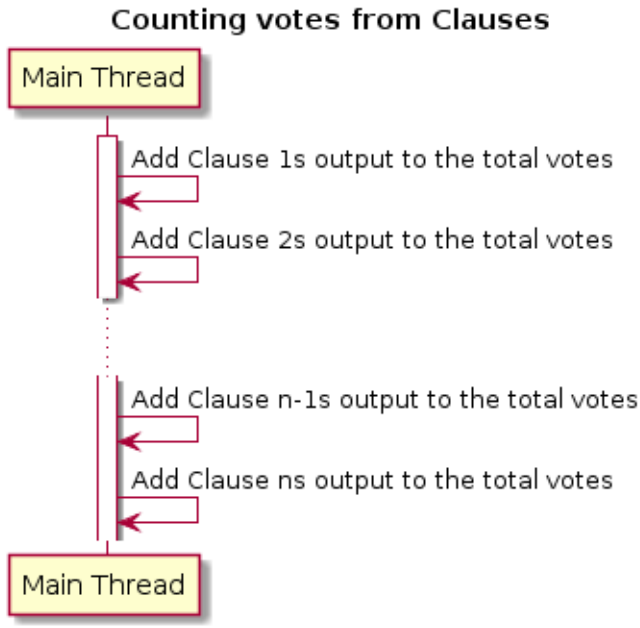

Figure 5.1: The current implementation of the Tsetlin Machine uses one CPU to count the output from each Clause in a Class.

5.2 Proposed solution

There are several ways of implementing a reduction algorithm in a GPU [38] [27]. In order to implement a parallel in the Tsetlin Machine, the polarity of a Clause needs to be addressed (votes for and against the class). Luckily, Equation 3.2 already has this mechanism implemented. Therefore, the proposed solution is to use parallel reduction with sequential addressing [27] combined with partial reduction [27]. There is one important requirement related to this approach. The amount of threads (processors) that execute the algorithm, needs to be a product of 2 (2^x). This is due to how the proposed solution loops through the outputs of Clauses, see Figure 5.2 and Figure 5.3.

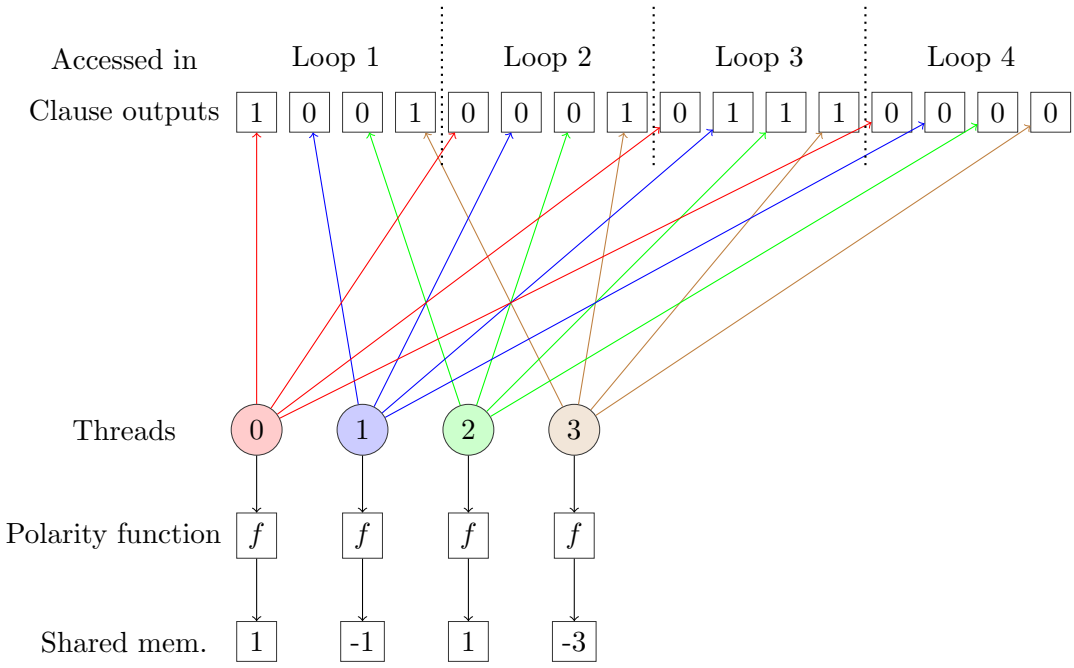

Figure 5.2: Multiple threads calculate a partial sum using a stridden (stride) loop through the array of Clause outputs. Finally, the output is stored in shared memory after being altered by the polarity of the Clauses.

First, the polarity of the Clause output needs to be calculated. This is achieved using a stridden parallel loop through the outputs from the Clauses. This technique allows the work to be distributed across several threads. An example of this is shown in Figure 5.2. In this example, there are 16 Clause outputs that are being processed by 4 threads. The total amount of work that needs to be performed by each of the threads can be performed in 4 loops. If it was only one thread, the total number of loops would have been 16.

Secondly, the task is to add all the numbers in the shared memory together. In Figure 5.3, an example of how that is done with the proposed solution is performed. This task is divided into 5 steps (since there are 2^5 sums in the shared memory).

I.e. in Figure 5.3, the sums are distributed in all the elements in the shared memory array, a total of 16 elements. First, threads from 0 through 7, retrieves

the score at their own index, and the score at an index of their own index + an offset of 8. Thus, thread 0 calculates the sum of the value in position 0 and 8. The output is stored in the cell with the same index as the thread id.

The next step is to reduce the offset and remove half of the threads. The amount of threads that will be used in the next step is $8/2 = 4$. In addition, the offset is divided by 2. Thereby, resulting in a new offset of 4.

This cycle is repeated until there is only one thread left. Then, the result from the reduction will be stored at index 0, as seen in Figure 5.3.

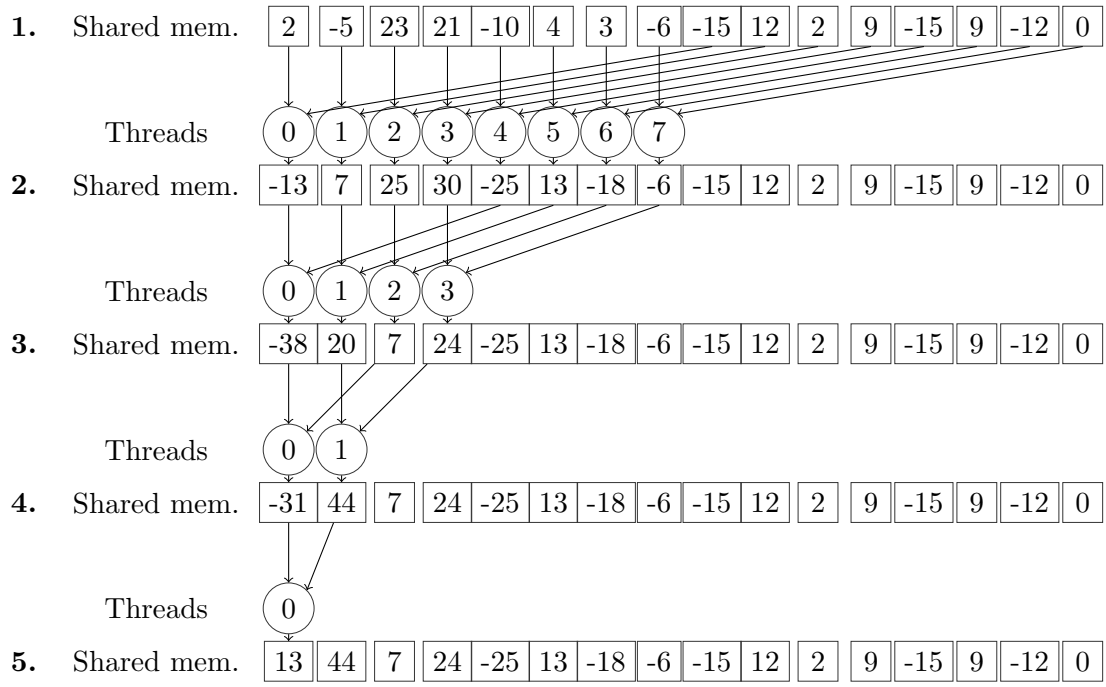

Figure 5.3: An example of how to reduce the partial sums from the first step using 8 threads and 16 sums.

5.3 Experiment

In order to test the CPU versus GPU reduction, an experiment was conducted. The experiment consisted of the same mechanics as of the Count votes phase in the Tsetlin Machine algorithm. Recall, in this phase, the Tsetlin Machine is to count the outputs from each Clause within a Class and apply a polarity on the sum. The following conditions are tested:

- The correctness of the procedure. In other words, if the results are correct.
- The execution time of the procedure.

The test was conducted 50 000 times to eliminate noise in the measurement, on 31 different dataset sizes. Ranging from 2^0 all the way to (including) 2^{30} number of Clause outputs. The intent was to see if the execution time was better on any of the two algorithms, and if so, find a guideline for when one of them was to be utilized over the other.

5.4 Results

The results from running the experiments 50 000 times and calculating the average execution time on each of the dataset sizes are shown in Table 5.1 [1].

In addition, a side by side plot of the results in Table 5.1, is shown in Figure 5.4.

[1] More detailed results and a Jupyter Notebook are available at this thesis's Github repository [44]

Dataset size	CPU	GPU
2^0	0.0000014041	0.0000122265
2^1	0.0000004752	0.0000071717
2^2	0.0000004665	0.0000070232
2^3	0.0000005544	0.0000071622
2^4	0.0000005090	0.0000065421
2^5	0.0000005031	0.0000066912
2^6	0.0000006107	0.0000066723
2^7	0.0000007293	0.0000066653
2^8	0.0000009789	0.0000066826
2^9	0.0000015052	0.0000067128
2^{10}	0.0000029674	0.0000067577
2^{11}	0.0000049448	0.0000069495
2^{12}	0.0000094619	0.0000074599
2^{13}	0.0000184260	0.0000084084
2^{14}	0.0000362890	0.0000103301
2^{15}	0.0000718008	0.0000141564
2^{16}	0.0001417443	0.0000217758
2^{17}	0.0002839911	0.0000378221
2^{18}	0.0005715443	0.0000703452
2^{19}	0.0011348340	0.0001332294
2^{20}	0.0022774922	0.0002570482
2^{21}	0.0045425669	0.0005043868
2^{22}	0.0090661810	0.0013593011
2^{23}	0.0181292382	0.0027688047
2^{24}	0.0363013888	0.0057922750
2^{25}	0.0727972486	0.0122280613
2^{26}	0.1459792105	0.0257625273
2^{27}	0.2925265660	0.0530825586
2^{28}	0.5862823975	0.1077091220
2^{29}	1.1773335444	0.2166417535
2^{30}	2.3725780540	0.4349740997

Table 5.1: Average execution time from running the reduction experiment 50 000 times on both CPU and GPU. Time is represented in seconds.

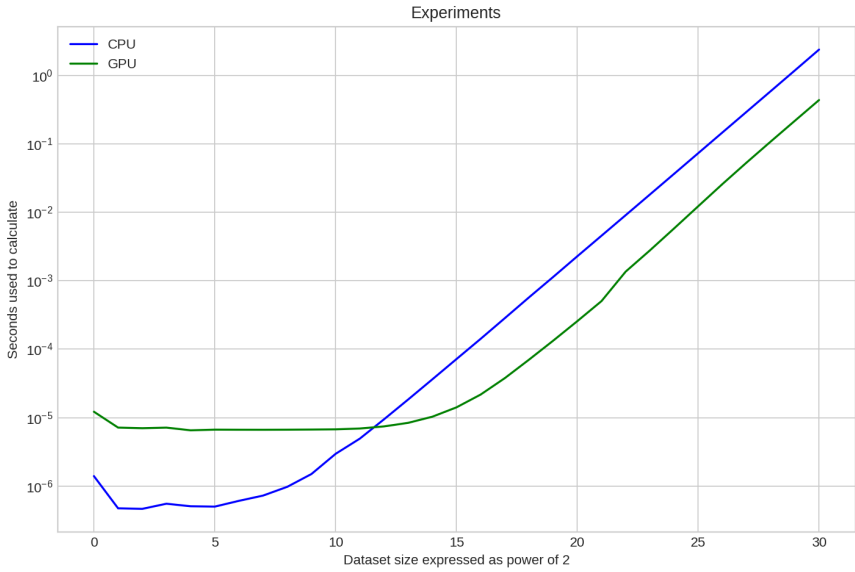

Figure 5.4: A comparison of the average results from CPU and GPU reduction from Table 5.1.

5.5 Conclusion

From the results, one can easily extrapolate when a CPU reduction pattern is to be utilized and when a GPU reduction pattern is to be utilized. It is clear that the power of the CPU outperforms the GPU on smaller datasets. However, when the number of elements to calculate grows beyond approximate 2^{12}, a GPU reduction pattern should be utilized to save execution time. This is clarified in Figure 5.4.

The reason why GPU is slower than the CPU on smaller datasets is probably due to the speed on each of threads in the GPU not being fast enough to compete with a much faster single core CPU. In addition to running on slower hardware, the reduction method on GPU brings some overhead that, when used on smaller datasets, becomes significant enough to increase the total execution time. However, on larger datasets, this overhead seems to almost disappear due to the computing time required to process that many elements.

This concludes Goal 4 of the thesis, where the goal was to investigate CPU versus GPU on counting votes. The results from this experiment show that there is a threshold for when to use CPU and when to use GPU when counting the votes with these proposed solutions.

Finally, regarding Hypothesis 2 with its given premise and the evidence provided in this chapter, it is safe to assume that this hypothesis is correct.

Chapter 6

Dynamic Parameters

One major drawback with the current implementations of the Tsetlin Machine is that all of the parameters need to be known during compile time [22] [20]. In a production environment, this is simply not an option. Often decisions need to be taken quickly by the system, and in other setups, the machine may not even have a compiler installed. To circumvent this problem, an implementation where these parameters could be set and even adjusted during runtime is proposed. The advantages of this approach are many [45], and include the following:

1. Adjusting the parameters do not require the code to recompile.
2. The parameters could be adjusted during the training.
3. Even changing the domain and model do not require a recompile.

Nonetheless, these advantages do not come for free. The most important drawback with this approach is that the code can never be as fast as hardcoded values [6]. This chapter is related to thesis Goal 5.

6.1 Dynamic S value

The first interesting value to look at in the Tsetlin Machine is the s parameter. This value is used to adjust how fast the training occurs. The higher this value

is, the slower the algorithm learns, while the lower the value is, the faster the algorithm learns. The difficult part is to find a balance between fast and precise learning [17].

By utilizing dynamic parameters, the Tsetlin Machine can update this value during training and even try to search for an optimal value to use. Using a technique called Stochastic Search On The Line (SSL) [57], the algorithm is able to search for a value that gives better accuracy than the previous value.

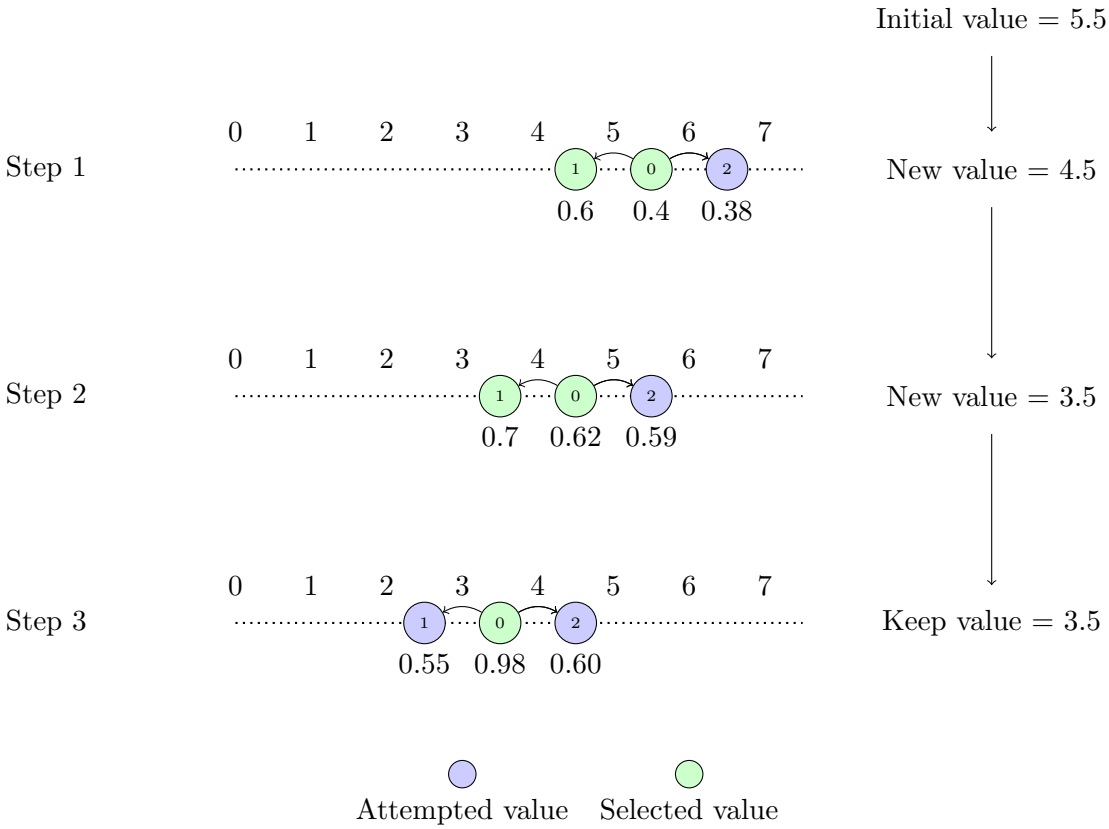

Figure 6.1: An example of how the s changes during training using SSL using an initial value of 5.5 and a d of 0.5.

The s value needs to be initially set to a decimal number. The amount to move (d) in a positive or negative direction is required as well. In Figure 6.1, an example of

the algorithm is divided into three steps. In the first step, an initial value of s is passed to the model and the accuracy of the model is measured to 0.4 with an s of 5.5. Thereafter, the s value is adjusted to 4.5 (decremented by d) and measured to 0.6. Lastly, an s value of 6.5 (incremented by d) is attempted and measured to an accuracy of 0.38. From these three results, the optimal result is to decrement the s value by d. The new s value is selected and stored as 4.5. These steps are repeated until the end of the training, as seen in Figure 6.1.

The parameters that need to be set in order for this technique to be applied are the following:

- s - An initial s value needs to be set in order for the model to have somewhere to start.
- d - The amount the s should move between the epochs.

In addition, it is required to know what action to try out. In order to calculate that, Equation 6.1 is proposed and utilized.

$$\text{Action} = \text{Epoch}_x \mod 3 \tag{6.1}$$

The Action could either be 0, 1 or 2. With that given Action, the model knows what the next value of s should be, as shown in Figure 6.1. Thereby, an Action of 0 would indicate that it should run with the current s value. An Action of 1, indicates that it should decrement the s by d and evaluate with that. Last, if an Action of 2 is given, the model should increment the s by d, and then evaluate the model.

The final adjustment of the model happens on Epochs where the Action equals 0, except for Epoch_0.

6.2 Experiment

In order to test the proposed dynamic s value, several tests are required. First, it is essential to see that the algorithm manages to adjust itself towards an optimal value. Thereafter, this test needs to be repeated for several scenarios.

In order to verify that the s approaches an optimal solution, we need a dataset that we know the optimal s value from. The Noisy XOR dataset [22], have already been tested in Tsetlin Machine paper, where the calculated optimal value is estimated to be 3.9 [16].

Several tests were required to verify if the s value is able to learn by itself:

1. **Experiment** - NoisyXOR dataset, with an initial s value of 15 and d of 0.5.
2. **Experiment** - NoisyXOR dataset, with an initial s value of 0.5 and d of 0.5.
3. **Experiment** - NoisyXOR dataset, with an initial s value of 15.0 and d of 1.0.
4. **Experiment** - NoisyXOR dataset, with an initial s value of 1.0 and d of 1.0.

6.3 Results

Following is the results of the experiments described in Section 6.2. A graph of all the experiments is shown in Figure 6.2. Next, individual graphs of each experiment, are shown in each subsection hereafter [1].

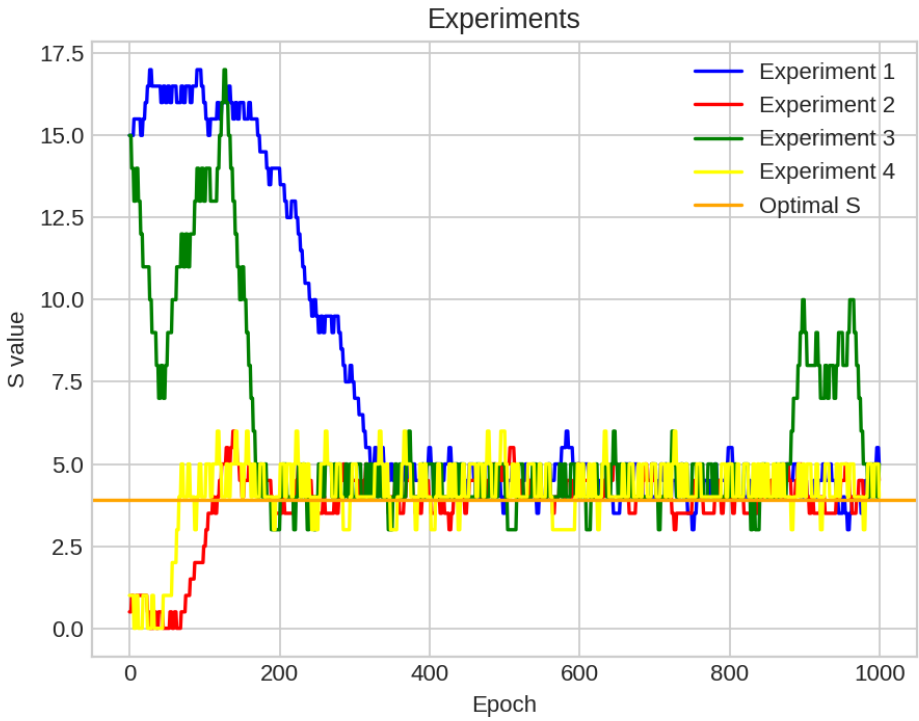

Figure 6.2: Plot from each of the experiments from the dynamic s value.

[1] More detailed results and a Jupyter Notebook are available at this thesis's Github repository [44]

6.3.1 Experiment 1

In Figure 6.3, the s starts out at 15.0 and after a while starts to adjust itself down towards the optimal s value. Once the s reaches 4.5 it starts to fluctuate around this position through the rest of the experiment.

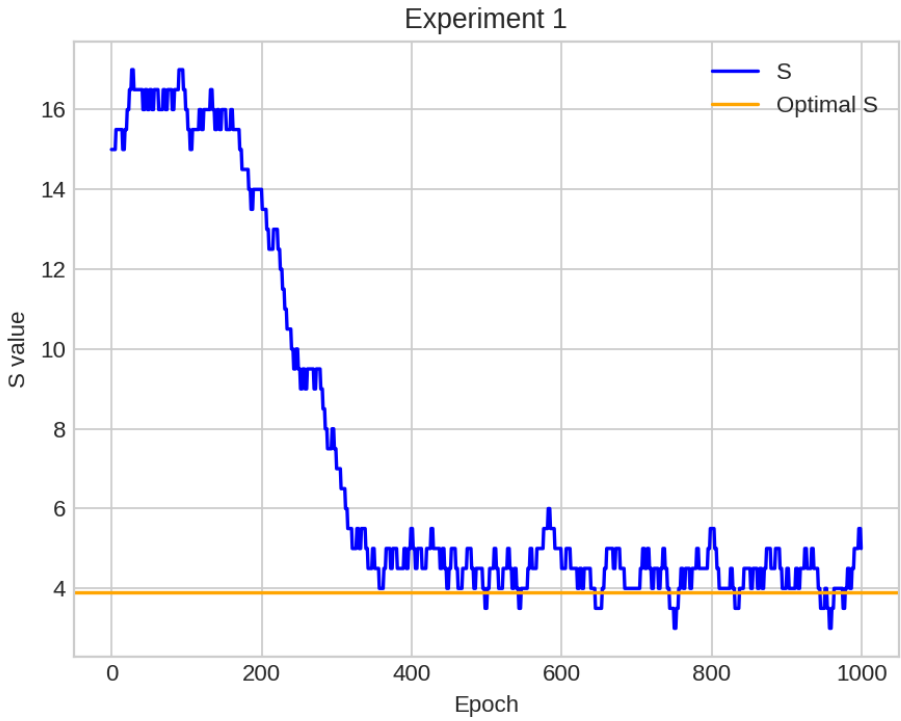

Figure 6.3: The s changes during the epochs for experiment 1.

6.3.2 Experiment 2

In Figure 6.4, the s starts out at 0.5 and after a while starts to adjust itself up towards the optimal s value. Once the s reaches 5.0 it starts to flat out and fluctuate around 4.5 and 3.5. through the rest of the experiment.

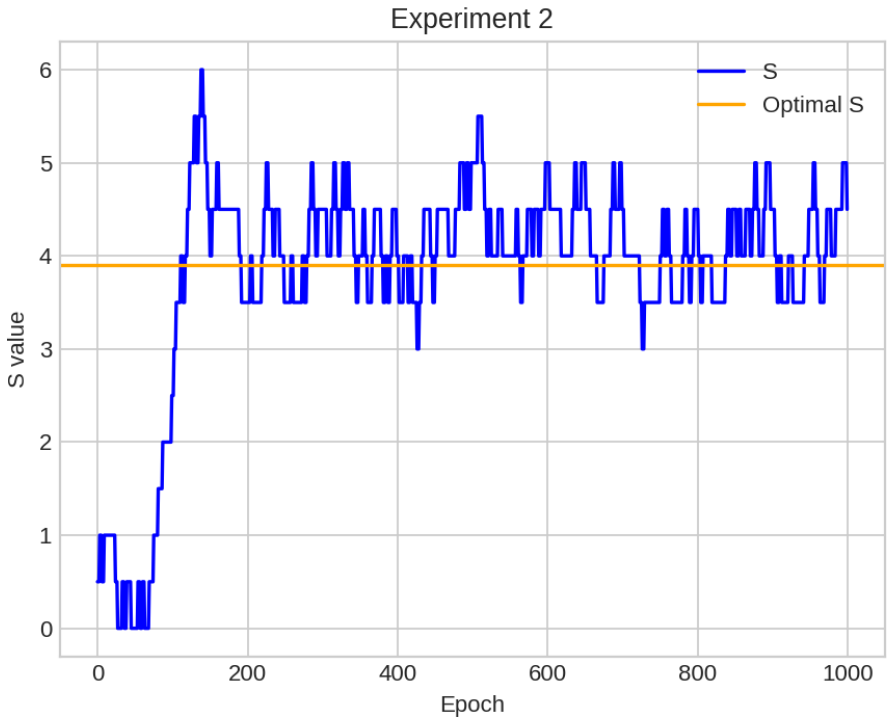

Figure 6.4: The s changes during the epochs for experiment 2.

6.3.3 Experiment 3

In Figure 6.5, the s starts out at 15.0 and after a while starts to adjust itself down towards the optimal s value. Once the s reaches 4.5 it starts to fluctuate around this position through the rest of the experiment.

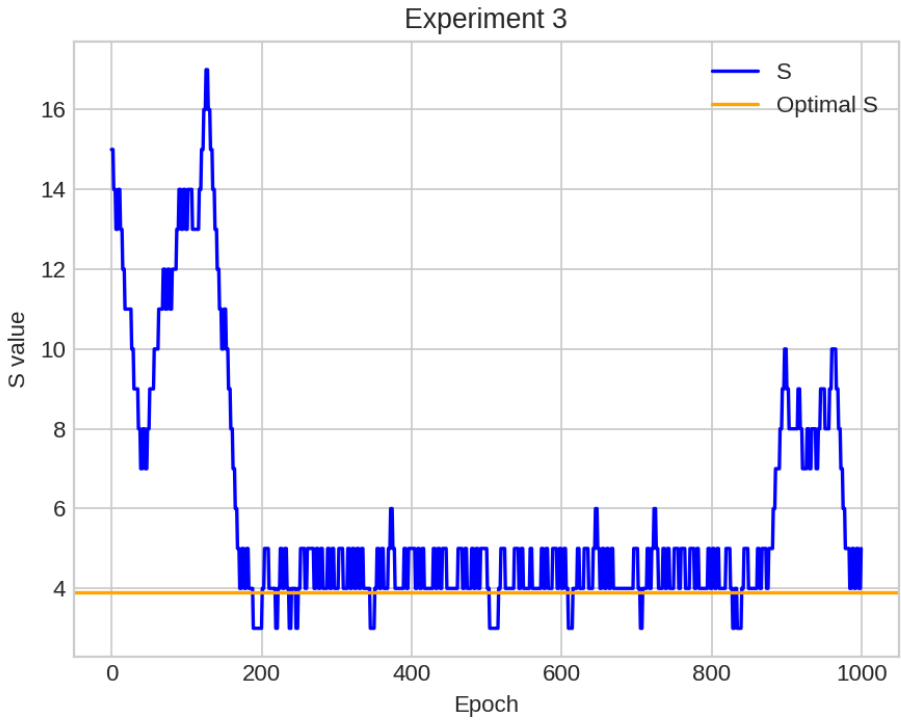

Figure 6.5: The s changes during the epochs for experiment 3.

6.3.4 Experiment 4

In Figure 6.6, the s starts out at 1.0 and after a while starts to adjust itself down towards the optimal s value. Once the s reaches 4.0 it starts to fluctuate around this position through the rest of the experiment.

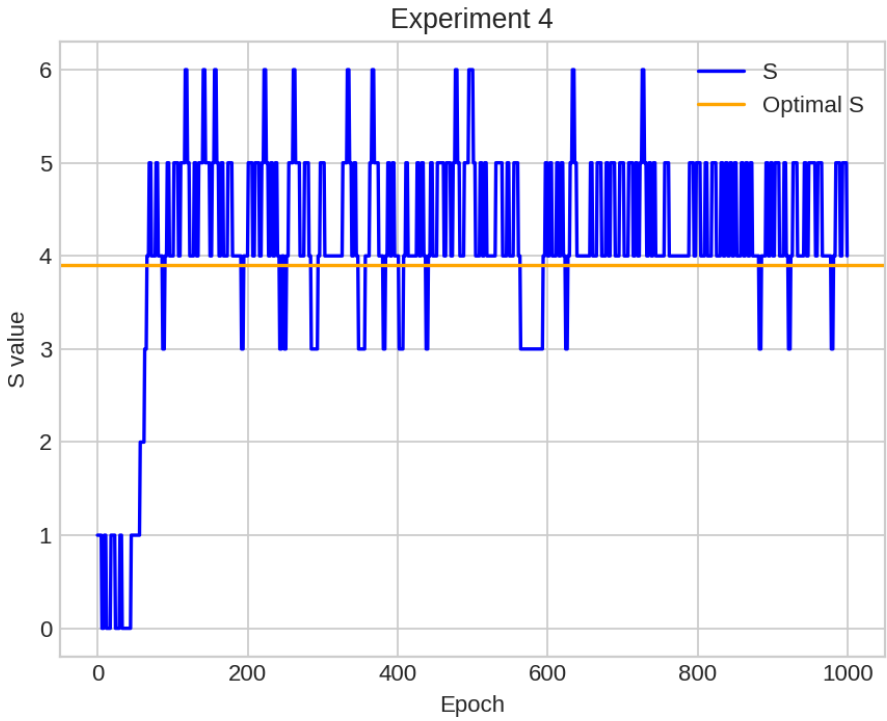

Figure 6.6: The s changes during the epochs for experiment 4.

6.4 Conclusion

As seen in Figure 6.2, the s value was able to adjust itself towards the known optimum value from the Tsetlin Machine paper [16]. Although in some cases it took many epochs before the adjustments, this could have been introduced by noise in training or the fact that the environment changes for each run. Nevertheless, with a primitive d, the model is still able to adjust the s to move closer towards an optimum value.

In the experiments, the known optimal s value was not reachable (since it was outside of the jumpable values). However, early in the testing, a too small d value did not allow the s value to move at all. Thereby, resulting in the s value staying at the same location. This could have been a consequence of the environment being too unstable to handle such small changes, making the accuracy changes from adjusting s, more or less random.

This concludes Goal 5 of this thesis, where dynamic parameters help the model to find optimal parameters. The experiments show that it is possible to find a good parameter, however, it might be difficult to pinpoint the exact best parameter. Nevertheless, that is not always required as long as the final results are good enough.

This test was conducted on only one dataset, it is expected that following the same procedure would yield almost the same results on another dataset.

With regards to Hypothesis 2 and the evidence provided in this Chapter, it is clear that the model is able to adjust and improve some of its parameters (s value) during training.

Chapter 7

Conclusion and Future Work

In this chapter, a thesis wide conclusion is presented that concludes all of the chapters respectively conclusions. In addition, some proposals for future work are given.

7.1 Conclusion

This thesis proposed a new architecture and investigated several aspects of the Tsetlin Machine algorithm. First, the Tsetlin Machine algorithm was introduced on an abstract level. Secondly, a proposed architecture for allowing the Tsetlin Machine to be executed in parallel on both CPU and GPU was suggested and tested. Thirdly, a solution for performing the reduction in order to count the votes in the Tsetlin Machine was tested on the CPU and GPU. Lastly, a solution for adjusting the s value during training was suggested using the stochastic search on the line [57].

In Chapter 3, Goal 1 was to provide an introductory description of the Tsetlin Machine algorithm. With the sample expansion and using logic gates, a proposed description was provided.

Further, in Chapter 4 and according to Goal 2, an architecture that allowed for parallelization of both CPU and GPUs was proposed. As a result, two experiments were conducted in accordance with Goal 3. They presented evidence that the proposed architecture works. Thus, utilizing more GPUs helps to reduce the total

training time. One interesting outcome in the results in Section 4.5, was the fact that the multi-threaded CPU implementation was faster than multiple GPUs. As mentioned, this could be the result of the datasets not being large enough that parallelism on GPUs was beneficial.

Next, in Chapter 5 and Goal 4, a technique called reduction, was introduced. This technique provided evidence for Hypothesis 1 to be true. Regardless, this raised some new questions, like when is the model large? However, if the algorithm is to handle larger models with larger datasets, it is without question a good idea to have a fast reduction method. Considering this is one of the vital parts of the algorithm.

In Chapter 6 and Goal 5, incorporated the Stochastic Search On The Line algorithm into the Tsetlin Machine algorithm. Consequently, allowing the algorithm to search for optimal hyperparameters during training. Just as important, this provided evidence for Hypothesis 2 to be true. The model showed signs of being able to learn a better parameter during training. In addition, it seemed to converge towards the known optimal value. As a result, providing the strongest evidence for Hypothesis 2 to be true.

The findings in this thesis open the door for many new paths for the Tsetlin Machine algorithm. Some being highlighted in the next section of this chapter.

7.2 Future work

In this section, future work is proposed and suggestions for what to improve or research, related to this thesis.

7.2.1 Synergy with Fast Tsetlin Machine

Even though not a part of this thesis, it is clear that a synergy approach between the proposed framework and the Fast Tsetlin Machine is all but inevitable. This would bring huge performance improvement to the algorithm, therefore, allowing it to handle even more complex problems that require more computational resources.

7.2.2 Python bindings

In order to move the Tsetlin Machine algorithm from the academic community into the industry, the right tools are required. Creating a connection from Python into this architecture allows developers with little knowledge for machine learning, to utilize the algorithm for their domain.

7.2.3 Implement Regression

Since this thesis was started, more abbreviations of the Tsetlin Machine has been introduced. One of them is the Regression Tsetlin Machine [1]. Investigating the possibility of implementing the regression Tsetlin Machine with this architecture, is something that is worth investigating. This would make the Tsetlin Machine more capable of solving new tasks.

7.2.4 Look at improved search mechanics for s parameter

From the experiments in Chapter 6, the s value was adjusted during training to find an optimal value. This however, could take many epochs and if the s value starts way of the optimal position, it might not be able to find it with a low d value. An improved mechanism for searching and testing out values, could enhance this feature.

7.2.5 Complete CUDA implementation

With the latest GPUs from Nvidia, comes new features. One of them being dynamic parallelism [2]. This concept, combined with Cooperative Groups [28] could be a natural next step for the algorithm. As a consequence, allowing the training process to run completely on one more GPUs.

References

[1] K. Darshana Abeyrathna, Ole-Christoffer Granmo, Lei Jiao, and Morten Goodwin. The Regression Tsetlin Machine: A Tsetlin Machine for Continuous Output Problems. *arXiv*, May 2019.

[2] Andy Adinets. CUDA Dynamic Parallelism API and Principles | Parallel Forall, May 2014. [Online; accessed 1. May 2019].

[3] Shameem Akhter and Jason Roberts. *Multi-core programming*, volume 33. Intel press Hillsboro, 2006.

[4] Blaise Barney. Introduction to Parallel Computing, Jun 2018. [Online; accessed 2. May 2019].

[5] Geir Thore Berge, Ole-Christoffer Granmo, Tor Oddbjørn Tveit, Morten Goodwin, Lei Jiao, and Bernt Viggo Matheussen. Using the Tsetlin Machine to Learn Human-Interpretable Rules for High-Accuracy Text Categorization with Medical Applications. *arXiv*, Sep 2018.

[6] Hadi Brais. Compilers - What Every Programmer Should Know About Compiler Optimizations, May 2019. [Online; accessed 3. May 2019].

[7] Adrian Bridgwater. An overview of deep learning tools, May 2019. [Online; accessed 1. May 2019].

[8] André R Brodtkorb, Trond R Hagen, and Martin L Sætra. Gpu programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*, 2012.

[9] André R. Brodtkorb, Trond R. Hagen, and Martin L. Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *J. Parallel Distrib. Comput.*, 73(1):4–13, Jan 2013.

[10] Siddhartha Chatterjee, Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 16–28, New York, NY, USA, 1993. ACM.

[11] François Chollet et al. Keras: Deep learning library for theano and tensorflow. *URL: https://keras. io/k*, 7(8):T1, 2015.

[12] Niklas Donges. Pros and Cons of Neural Networks. *Towards Data Science*, Jul 2018.

[13] Luke Dormehl. What is an artificial neural network? Here's everything you need to know. *Digital Trends*, Jan 2019.

[14] Jocelyn D'Souza. An Introduction to Bag-of-Words in NLP. *Medium*, Jun 2018.

[15] GeeksForGeeks. Queue Data Structure - GeeksforGeeks, May 2019. [Online; accessed 3. May 2019].

[16] Ole-Christoffer Granmo. The Tsetlin Machine - A Game Theoretic Bandit Driven Approach to Optimal Pattern Recognition with Propositional Logic. *arXiv*, Apr 2018.

[17] Ole-Christoffer Granmo. Private communication during supervision sessions, 2018/2019.

[18] Ole-Christoffer Granmo. cair/fast-tsetlin-machine-in-cuda-with-imdb-demo Repository, May 2019. [Online; accessed 2. May 2019].

[19] Ole-Christoffer Granmo. cair/fast-tsetlin-machine-with-mnist-demo Repository, May 2019. [Online; accessed 2. May 2019].

[20] Ole-Christoffer Granmo. cair/TextUnderstandingTsetlinMachine Repository, May 2019. [Online; accessed 2. May 2019].

[21] Ole-Christoffer Granmo. cair/TsetlinMachine Repository, May 2019. [Online; accessed 2. May 2019].

[22] Ole-Christoffer Granmo. cair/TsetlinMachineC Repository, May 2019. [Online; accessed 2. May 2019].

[23] Paul R. Daugherty H. James Wilson and Chase Davenport. The Future of AI Will Be About Less Data, Not More, Jan 2019. [Online; accessed 11. May 2019].

[24] Mark Harris. An Easy Introduction to CUDA C and C++, Oct 2012. [Online; accessed 3. May 2019].

[25] Mark Harris. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops | NVIDIA Developer Blog, Apr 2013. [Online; accessed 3. May 2019].

[26] Mark Harris. GPU Pro Tip: CUDA 7 Streams Simplify Concurrency, Jan 2015. [Online; accessed 3. May 2019].

[27] Mark Harris et al. Optimizing parallel reduction in cuda. *Nvidia developer technology*, 2(4):70, 2007.

[28] Mark Harris and Kyrylo Perelygin. Cooperative Groups: Flexible CUDA Thread Programming | NVIDIA Developer Blog, Oct 2017. [Online; accessed 1. May 2019].

[29] Robert Hecht-Nielsen. III.3 - Theory of the Backpropagation Neural Network**Based on "nonindent" by Robert Hecht-Nielsen, which appeared in Proceedings of the International Joint Conference on Neural Networks 1, 593–611, June 1989. © 1989 IEEE. *Neural Networks for Perception*, pages 65–93, Jan 1992.

[30] Patrick Hohenecker and Thomas Lukasiewicz. Ontology Reasoning with Deep Neural Networks. *arXiv*, Aug 2018.

[31] National Instruments. Teaching Digital Logic Fundamentals - Theory, Simulation and Deployment - National Instruments, Oct 2013. [Online; accessed 28. Apr. 2019].

[32] Anil K Jain, Jianchang Mao, and KM Mohiuddin. Artificial neural networks: A tutorial. *Computer*, pages 31–44, 1996.

[33] Christopher Williams Andrew Hsu Jimin Khim John McGonagle, George Shaikouski. Backpropagation | Brilliant Math & Science Wiki, May 2019. [Online; accessed 3. May 2019].

[34] Srinidhi Kestur, John D. Davis, and Oliver Williams. BLAS Comparison on FPGA, CPU and GPU. *IEEE*, pages 5–7, 2019.

[35] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[36] Shahar Kvatinsky, Guy Satat, Nimrod Wald, Eby G. Friedman, Avinoam Kolodny, and Uri C. Weiser. Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies. *IEEE Trans. Very Large Scale Integr. VLSI Syst.*, 22(10):2054–2066, Oct 2014.

[37] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436, May 2015.

[38] Justin Luitjens. Faster Parallel Reductions on Kepler, Feb 2014. [Online; accessed 2. May 2019].

[39] Laura McDonald, Sreeram V. Ramagopalan, Andrew P. Cox, and Mustafa Oguz. Unintended consequences of machine learning in medicine? *F1000Research*, 6, 2017.

[40] Vivian Zhang Neimeth and Chris. Why data science and machine learning are the fastest growing jobs in the US. *InfoWorld*, Mar 2018.

[41] Michael A. Nielsen. Neural Networks and Deep Learning - How the back-propagation algorithm works. *Determination Press*, 2015.

[42] Michael A. Nielsen. Neural Networks and Deep Learning - Using neural nets to recognize handwritten digits. *Determination Press*, 2015.

[43] Nvidia. CUDA C Programming Guide, Mar 2019. [Online; accessed 3. May 2019].

[44] Anders Refsdal Olsen. andersro93/master-thesis-source Repository, May 2019. [Online; accessed 17. May 2019].

[45] P. Oreizy, N. Medvidovic, and R. N. Taylor. *Architecture-based runtime software evolution.* IEEE, Apr 1998.

[46] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU Computing. *Proc. IEEE*, 96(5):879–899, May 2008.

[47] A Roques. Plantuml: Open-source tool that uses simple textual descriptions to draw uml diagrams, 2015.

[48] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5):206–215, May 2019.

[49] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[50] Christos Stergiou and Dimitrios Siganos. Neural Networks, May 1997. [Online; accessed 2. May 2019].

[51] Marc A. Suchard, Quanli Wang, Cliburn Chan, Jacob Frelinger, Andrew Cron, and Mike West. Understanding GPU Programming for Statistical Computation: Studies in Massively Parallel Massive Mixtures. *Journal of Computational and Graphical Statistics*, 19(2):419–438, Jan 2010.

[52] TensorFlow. TensorFlow, May 2019. [Online; accessed 2. May 2019].

[53] Axel Tidemann. nevralt nettverk – Store norske leksikon, May 2019. [Online; accessed 11. May 2019].

[54] M. L. Tsetlin. FINITE AUTOMATA AND MODELS OF SIMPLE FORMS OF BEHAVIOUR. *Russ. Math. Surv.*, 18(4):1–27, 1963.

[55] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH computer architecture news*, volume 23, pages 392–403. ACM, 1995.

[56] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Comput. Sci. Eng.*, 13(2):22–30, Mar 2011.

[57] Anis Yazidi, Ole-Christoffer Granmo, and B John Oommen. A stochastic search on the line-based solution to discretized estimation. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 764–773. Springer, 2012.

Appendices

A Hardware Specification

All the tests was conducted on a Nvidia DGX-2 server with the following specifications.

Operating System	Ubuntu 18.04 LTS
Processor	Intel Xeon Platinum 8168 x4 @ 24 C (96 C in total) 2.7GHz
Memory	1.5 TB DDR4
Graphics	16x Tesla V100-SXM3-32GB with NVLINK

UiA University of Agder
Master's thesis
Faculty of Engineering and Science
Department of ICT