VISUALISATION OF THE TSETLIN MACHINE

BY

REUBEN ATHERTON

NEWCASTLE UNIVERSITY SCHOOL OF ENGINEERING

SUPERVISORS: RISHAD SHAFIK, ALEX YAKOVLEV

Contents

I. Abstract

This paper discusses the value of machine learning visualisation, more specifically visualisation of the Tsetlin Machine. The motivation for such work includes how interpretability and visualisation of algorithms can be used to justify predictions made by machine learning models in high stake environments. Other motivations include the benefits of visualisation when studying new algorithms, as well as using visual tools to provide insight into the internal dynamics of the model, allowing for performance improvements to be made. The work associated with this paper uses visual driven experiments to:

a) demonstrate how class voting is implemented – this concept can be scaled up and used to help when learning different concepts within the TM.

b) plot the average number of TA state flips where a thesis was drawn such that the average rate of change of number of TA flips is inversely proportional to the learning rate 's' hyper-parameter. This thesis would not have been made without the visual aid of graphs plotted, further solidifying the benefits of such tools.

c) finally, propose a novel idea involving locally stochastic clause dropping where methods of data extraction for said idea are shown. Again, visualisation was implemented in order to get such results.

II. Introduction

Emerging from the burgeoning machine learning (ML) sector are increasingly accurate learning algorithms with powerful computational abilities, most of which have become too complex for humans to understand. These incomprehensible learning dynamics have led to an absence of interpretability, raising ethical concerns regarding the lack of understanding in the algorithm's decision-making process. An inability to rely on the model's predictions, regarding real world data, undermines the value of having such technologies at our disposal, making confidence within the system paramount. ML is used in many high-stake environments with examples of this being medicine or law. For instance, Artificial Neural Networks (ANNs) and Natural Language Processing (NLP) algorithms are used to create paradigms in unstructured medical data. In such cases, the people acting upon these predictions should be able to understand the motivations behind such decisions. Visualisation of said decision-making can help to understand the reasoning behind certain choices and to what extent the result is trustworthy, especially within these precarious fields of work. There have also been studies to suggest visualisation can assist in online training processes of deep learning models, further improving the veracity of the results [1].

Another notable necessity when preparing an ML model, is its real-world evaluation. The behaviour of these models can vary between testing on metrics of interest versus real data. The reason being, real-world data often varies significantly from that of emulated data, hence the model's performance may differ from what is expected following deployment [2]. Again, a deeper understanding of the models' internal dynamics allows for necessary changes to be made, with the intention of mitigating these performance differences, whilst also maintaining machine fidelity.

Looking at the lack of interpretability from another point of view, visualisation also makes sense with regards to teaching and studying ML algorithms. At first glance, these networks can appear seemingly

impossible to understand. Although this will always be the case, as with any complex theory, the learning process can be relieved by graphics and interactive real-time demonstrations.

Introducing the Tsetlin Machine (TM) [3], a developing solution to the black box ML approach. Using propositional logic and supervised learning to formulate clauses, we can better understand how these decisions are made [4]. This project will look at visualising the decisions made by the TM as well as the average number of flips between TA states after adjusting certain parameters within the initialisation of the TM. Further included within the scope of this project is consideration of how certain clauses could be withdrawn where there is evidence of a large difference in clause performance, in order to reduce computational load whilst preserving optimal accuracy.

1. Aims and Objectives

Aims lettered, goals (**G**) identified, and objectives are listed below.

a) To establish an in-depth knowledge of the TM's architecture, with a particular focus on clause computation and class voting **(G1)**.

b) To become comfortable with the TM code implementation, using different datasets (**G2**). This involves:
   i) Testing different Tsetlin Machines that are readily available on GitHub using different datasets to analyse behaviour.
   ii) Comparing different language implementations such as Python, C, Java etc. in order to collect similarities to solidify existing theoretical understanding.

c) Eventually create a visualisation tool, built upon an existing TM of a chosen language, displaying clause value summation and class voting **(G3)**. This involves:
   i) Identifying necessary variables that can be extracted from relevant functions, to be used and tested against different ideas for visualisation.
   ii) Performing individual logic expressions in order to breakdown the clause computation, outside the program's libraries.

d) Plotting the average number of TA flips between include and exclude across all 10 classes for the MNIST dataset, analysing how this number of flips changes when adjusting 's' and 'T' parameters **(G4)**. This involves:
   i) Extracting the number of TA flips from existing TM during training.
   ii) Building a script that will calculate and plot the average number of flips between all the 10 classes.
   iii) Running experiments on the TM for different values of 's' and 'T'.

e) Proposing a way in which visualisation could help move the TM forward **(G5)**. This involves:
   i) Examining current literature and identifying TM improvements that could be made as a result of visualisation and the insight provided.

Aims and objectives that are less related to the TM include:

a) Improving Python and C/C++ programming skills by analysing code and writing new scripts in order to analyse data **(G6)**.

b) Developing creative skills where an idea is presented and then built from scratch, making the plan come into fruition and eventually become something useful **(G7)**.

2. Paper Organisation

Section I of this paper aims to highlight motivations behind developing TM visual tools. Section II delivers a current understanding based on literature published within the field of ML and TM concepts alike. Section III provides technical theory regarding relevant TM internal learning dynamics. Section IV looks at the project specific results including that of a visualisation tool for the flipping TA states whilst also presenting new ideas for future visual developments. Finally, Section V concludes the paper.

III. Literature Review

1. Tsetlin Machine Comparison

Artificial Neural Networks (ANNs) are recognised for their governing role in building the future. Much like the human brain, ANNs employ the use of neurons as the basic unit of computation, receiving inputs from other neurons or external sources attached to weights expressing their relative importance, seeking to generalise a pattern [5]. In recent years however, the low-complexity TM has proved to compete against these neural networks (NNs) in areas such as learning convergence. The TM also boasts an energy efficiency of up to 15 times greater than that of the ANNs. With that being said, this vast betterment in energy efficiency comes at the cost of an increased energy consumption per epoch [6]. Another comparison to be made is the level of understandability present within these developing machine learning algorithms. Whilst ANNs provide no substantial explanation for predictions, the TM is its inherently interpretable, using rule-based algorithms to present more logical reasoning. With this comes a naturally greater reliability of the model's prediction.

2. The Importance of Interpretability

As previously mentioned, high accuracy predictions made by ML models are not enough on their own. In specific, less-consequential cases, the prediction released from the model is the only concern, however problems of greater complexity require transparency into the model's decision which is frequently achieved at the cost of machine accuracy. This trade-off provides a deeper human understanding of the problem, helping us study ways in which the model could behave erroneously and how to improve certain aspects [7]. Research into ML interpretability has gained a lot of attention over the recent years, inciting various steps taken towards achieving this greater level of apprehension that is so deeply sought after [8]. A lack of interpretability can allow for instances in which unidentified bias can sway predictions, where either the bias is inherently held by researchers and as such reflects in the models' algorithm or even in the data used for developing such models [9]. There are two approaches towards mitigating such circumstances including model-based-intrinsic interpretability and post-hoc-extrinsic interpretability. The first occurs during the running of the model, allowing a developed understanding of how to machine approaches training convergence. The latter is only implemented after the model runs [10] and provides an evaluation of the model. In the case of this project, the first proposed visualisation tool would fall under post-hoc-extrinsic interpretability, whilst the second is classified as model-based-intrinsic interpretability.

3. TM Interpretability and Weighted Clauses

The TM's native interpretability comes from its ability to build highly discriminative conjunctive clauses in propositional logic, acting as patterns within data enabling human inference. Modified learning mechanisms have been proposed whereby false positives are opposed, whilst true positives are encouraged. Consequently, there is an increase in clause discrimination power due to certain

weights being applied [11]. This paper presents a Weighted Tsetlin Machine (WTM), which was found to reduce the computation time as well as the memory usage. More recently, following the WTM's success, a similar idea was proposed to address the accuracy-interpretability challenge. An Integer Weighted TM (IWTM) was proposed, working by identifying weaker clauses and combining them in order to generate a higher accuracy as a group. This instance would resemble that of a lower weighted clause in the previous method. In opposition, more successful, accurate clauses are rewarded with a greater level of independence, expecting that they will continue to operate well on their own. These clauses would relate to the higher weighted clauses previously mentioned. This study shows how the trade-off between interpretability and accuracy can be enhanced, resulting in the use of 6.5x fewer literals than a standard TM and 120x fewer literals than a real-value weighted TM [12]. As a result, the IWTM demonstrated an overall improvement in precision and a greater level of human inference. Although the method of replacing multiple clauses with that of a single clause improved the overall performance of the TM, there were still no real-time charts produced that could demonstrate the formation of the clauses throughout training. This could be one of the aims for a future project within the scope of visualisation.

4. Stochastic Clause Dropping Implementation

Within the early ML models, interpretability was a naturally occurring asset, however with the increasing complexity adopted by most modern approaches, this later became less true [13]. The more understandable linear regression models are disadvantaged with less accuracy whilst the deep learning black box methods have greater accuracy in prediction, although prove to be less interpretable. With interpretability transpires a greater understanding of the problem and hence an opportunity for improvement. One issue with the current architecture of the TM is overfitting within the learning element. Since clauses are susceptible to capturing noise and irrelevant patterns, performance is negatively affected. Upon reducing the number of clauses, the reduction in the TM's performance was too sizeable to accept. An experiment to mitigate overfitting was then carried out, following inspiration from the dropout theory, originally implemented with NNs [14]. This experiment involved dropping a set of clauses stochastically during training. The results demonstrated a 10% increase in accuracy as well as 4x faster learning speed directly caused by the reduction in the number of clauses. There was also an improvement in interpretability [13].

5. Proposed TM Developments

Another element that is within the scope of this project is to observe the clause outputs for the correctly voted class and collect all the least useful clauses from that class. Where there are underperforming clauses, relative to other clauses within the same class, these clauses can be removed randomly. This presents a concept of locally random clause removal. For example, if a class is voted for upon the basis of a marginal vote, a random clause will be dropped from a collective group of the least useful clauses within that class. General reasoning would suggest that the consequences of such methods would provide similar results to the stochastic clause dropping scheme proposed by Granmo, where in this instance, the clauses dropped are more targeted whilst remaining locally random.

IV. Theoretical Background

1. TM Input

The TM takes the input Boolean vector X = $[x_0, \dots, x_n]$, where n is number of features. The literal set of the input is therefore given by $\mathcal{L} = \{x_0, \dots, x_n \dots, \neg x_0, \dots, \neg x_n\}$ where $\neg x_0$ is the negated literal, meaning '*not* x_0'. For example, the literals for a Noisy XOR input with 2 elements of noise would take

the form X = [0,1,1,0] meaning $\mathcal{L} = \{0,1,1,0,1,0,0,1\}$ where $\mathcal{L}$ has been split into literals and negated literals.

2. The Tsetlin Automaton

The Tsetlin Automaton (TA) is a new solution to the multi-armed bandit problem, in which there are two actions - include or exclude. The goal is to dynamically allocate the correct literals/negated literals, to a state resulting in the highest probability of receiving a reward. This can involve demoting unnecessary literals/negated literals to a state such that they will be excluded from the clause, also resulting in a reward. The action of rewarding always promotes confidence of the current state, as can be seen in Fig 1.0. Penalty actions always demote the confidence of the existing state and so move the TA state towards the middle (state boundary). For a token to flip between actions, there must be a penalty applied. Each literal is fed into its own TA where its 'relevance' is implemented. If included, the literal will be involved within the clause - one clause associates with a 'team' of TAs and there is one TA assigned per literal.

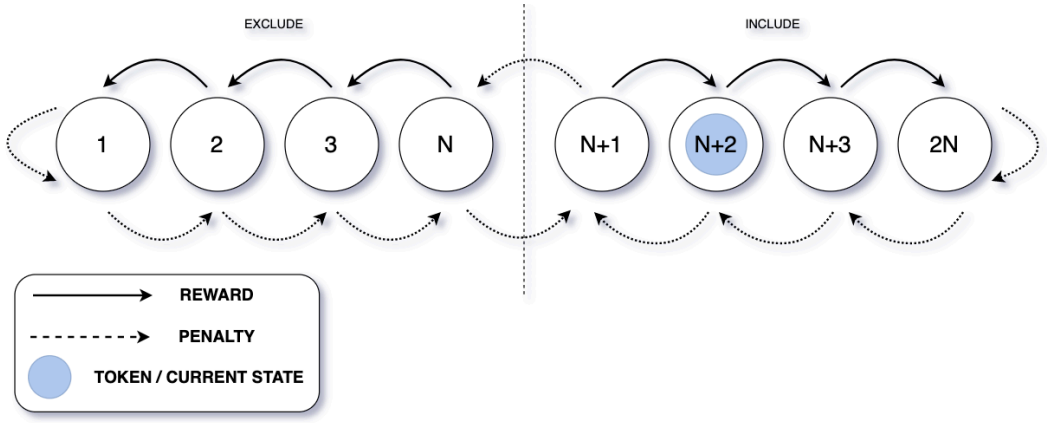

Fig 1.0 Tsetlin Automaton state diagram with include/exclude actions.

This fixed finite state method performs optimally in stochastic environments and requires low computational complexity. Due to the simplicity of the increment/decrement functionality, the TA only has a small memory footprint [4].

3. Clause Computation

Any literals deemed necessary (included) in that specific clause are then combined at a single point called the conjunction operator. Here, the TA outputs are ANDed together to produce the final clause taking the form:

$$\mathbf{C_j^i = 1 \wedge \left(\wedge_{k \in I_j^i} x_n\right) \wedge \left(\wedge_{k \in \bar{I}_j^i} \bar{x}_n\right)}$$

where $\boldsymbol{I_j^i}$ are the included literals and $\boldsymbol{\bar{I}_j^i}$ are the included negated literals. Take for example the Noisy XOR input of X = [0,1,1,0] with the array elements indexed as X = $[x0, x1, x2, x3]$. Using an online interpretability tool [15], once trained, the TM creates a user defined number of clauses, one of which being $\boldsymbol{C_j^i = x1 \wedge \neg x0}$, meaning that the TM has decided these literals are essential to the clause. Upon substituting in the literal values, this clause gives the following output:

$$\boldsymbol{C_j^i = x1 \wedge \neg x0}$$
$$\boldsymbol{C_j^i = 1 \wedge x1 \wedge 1 \wedge \neg x0}$$

$$C_j^i = 1 \wedge 1 \wedge 1 \wedge 1$$
$$C_j^i = 1$$

To summarise the creation of clauses, Fig 1.1 is a flow chart to show the clause cycle, starting with the input data, ending with the clause itself.

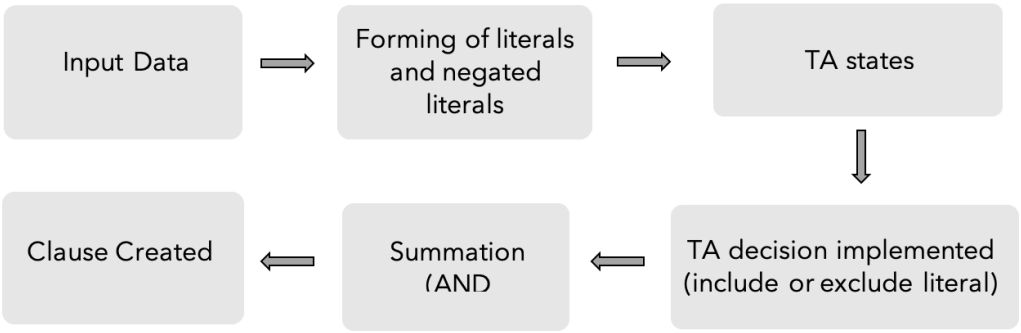

Fig 1.1 A flow chart to show the creation of clauses within the TM

4. Class Voting

When the TM votes, it chooses which class is best suited to the input it has been provided. In the case of the Noisy XOR, there are only two classes, $\boldsymbol{Y} = [\mathbf{0}, \mathbf{1}]$ due to its intrinsic Boolean nature. In the case of the MNIST dataset there are ten classes, $\boldsymbol{Y} = [\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots, \mathbf{9}]$ where each class is a possible prediction relating to certain inputs. Each class is comprised of several clauses, the number of which is user customisable. They are then split into positive and negative halves. Usually, the odd clauses are assigned a negative polarity and the even clauses assigned a positive polarity. Looking at all the clauses generated by the interpretability tool [7], in the case of the Noisy XOR dataset for the class **Y = 0**, the following result is produced:

Class 0: Positive (+)

#0: **x0 ∧ x1 = 0 ∧ 1 = 0**
#2: **x0 ∧ x1 = 0 ∧ 1 = 0**

Class 0: Negative (-)

#1: **x1 ∧ ¬x0 = 1 ∧ 1 = 1**
#3: **x1 ∧ ¬x0 = 1 ∧ 1 = 1**

These polarities determine the vote for that class, either 'vote for' (positive polarity) or 'vote against' (negative polarity). The actual vote is decided by summation and thresholding.

5. Summation and Thresholding

With the two halves of the class initialised with their respective polarities, they are they subject to summation. The following equations give the class confidence, u:

$$u = \sum_{j=1}^{\frac{n}{2}} C_j^+(X) - \sum_{j=1}^{\frac{n}{2}} C_j^-(X)$$

The above equation can be simplified to give:

$$\mathrm{u} = \text{positive clause values} + \text{negative clause values}$$

In the Noisy XOR example used throughout, the clauses of negative polarity add to give 2. A 'negative sign' is given to this value due to its predetermined negative polarity and the so the value is now -2. The positive half of clauses sums to give 0. Now calculating 'u' gives $u = 0 + (-2) = -2$.

The next step is thresholding. If $u < 0$, the class votes against itself and if $u \geq 0$, the class votes for itself. With a 'u' value of -2, the class will vote against Y = 0 (because this example concerns Class **Y = 0**) and again, due to the Boolean nature of the dataset, this essentially means vote for **Y = 1**. We know this prediction to be correct because the first two elements of the input are 0 and 1, producing an XOR result of 1.

6. Include/Exclude Logic

In the case of a human observing the label 'include' next to a literal when deciding what to involve in the clause, they would recognise that this specific literal is to be included. A machine however must use Boolean algebra. The way this is implemented is as follows:

Using a new two-input XOR of 0,1 where the negated literals are 1,0.

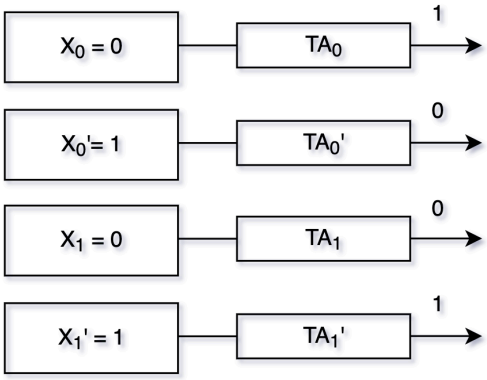

Fig 1.2 An example of literals and their connected TA states where 1 = include, 0 = exclude

Figure 1.2 shows 2 literals and 2 negated literals connecting to their respective TAs where the TA state has been pre-determined, shown by 'include action' depicted by 1 and 'exclude action' depicted by 0. The apostrophised, dashed variables are the negated literals. To implement these actions, Boolean algebra is implemented.

$$= (X_0 \vee \overline{TA_0}) \wedge (\overline{X_0} \vee \overline{TA'_0}) \wedge (X_1 \vee \overline{TA_1}) \wedge (\overline{X_1} \vee \overline{TA'_1}) \quad (1)$$

$$= (X_0 \vee 0) \wedge (\overline{X_0} \vee 1) \wedge (X_1 \vee 1) \wedge (\overline{X_1} \vee 0) \quad (2)$$

$$= (0 \vee 0) \wedge (1 \vee 1) \wedge (0 \vee 1) \wedge (1 \vee 0) \quad (3)$$

$$= 0 \wedge 1 \wedge 1 \wedge 1 \quad (4)$$

$$= 0 \quad (5)$$

This method utilises the ANDing (∧) and ORing (∨) of the literals with the negated version of the TA state and by doing so, it is the Boolean value of the literals that determines the eventual result of the clause. TA_0 is shown to be included in the clause (state = 1). Simply ORing this TA with the literal value would always result in a 1 being produced due to the nature of the OR gate; even if the literal

value was 0, the output would be 1 ($1 \wedge 0 = 1$). To bypass this, all literals are ORed with their respective negated TA state, switching all 'includes' (or 1s) to 'excludes' (or 0s). This gives the power of influencing the result back to the literal. This method also prevents any excluded literals with values of 1 from overpowering the OR operation. Each result is then ANDed together once again meaning that even one included literal-TA combination producing a 0, would result in the total clause value being 0, as shown in the above equation display.

V. Results and Discussion

1. Visualisation of Noisy XOR Voting

One of the original goals of this project was to create a tool with the ability to train a TM, gather the clauses created and then cross-check these clauses with a new, unseen input provided by the user to finally create a plot displaying how the voting for the classes was being fulfilled. The below Figures 1.3 to 1.6 are the results of such a tool. Firstly, the TM Interpretability Tool, available on GitHub [16], was first used to train with the Noisy XOR dataset, where both the training and testing data included a total of 5000 inputs, each containing 12 features at 69.8% signal to noise ratio. In each class, 10 clauses were used, divided up into their respective polarities such that even clauses (including clause 0) were assigned positive polarity and odd clauses assigned negative polarity. The accuracy of the TM was 100% which was to be expected for such a simple dataset. Once the TM was trained, the interpretability tool prints each clause for each class. The task then was to develop a deconstructing tool that could take the new input created by the user and cross check these literals included within the clause against the index of the new input. A Python script was built to achieve this where first the clauses are each appended to a new list, depending on their polarities. A parsing function was implemented that reads the index of each literal within these lists and returns the sum of that class's polarity. All four class totals were then summed together, as described in Section III. (5. Summation and Thresholding) to produce the class vote. The individual positive and negative class sums were then plotted onto a simple graph. In the case that the class shows a negative bar, this means the TM votes against this class and hence when a positive bar is produced, the TM votes for that class. As such every plot should produce two bars, one in the negative section and the second in the opposite class's positive section, demonstrating a clear, unopposed vote.

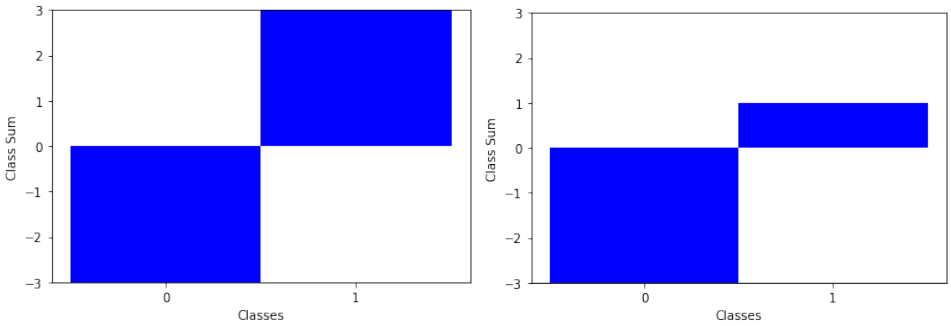

Fig 1.3 and Fig 1.4 show the results of the visualisation tool voting for Class 1 and against Class 0

Unseen Input array = [1,0,1,1,0,0,0,0,0,1,0]

Figures 1.3 and 1.4 show the TM voting for the class **Y = 1**, where the class **Y = 0**, is shown to be negative and therefore voted against. There is a difference in confidence indicated by the magnitude of each class sum, with the left figure displaying a more confident vote.

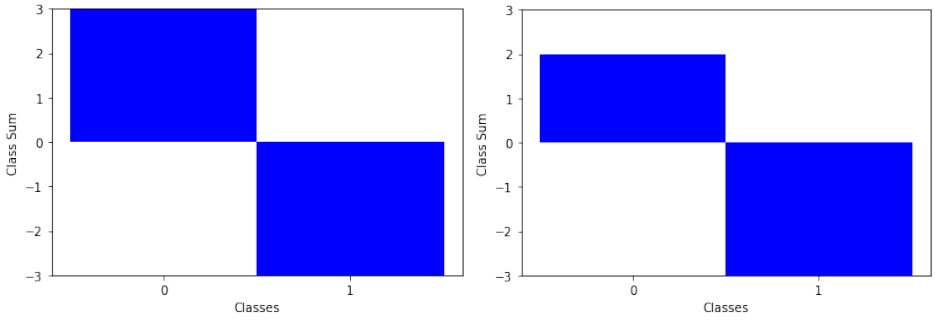

Fig 1.5 and Fig 1.6 show the results of the visualisation tool voting for Class 0 and against Class 1

Input array = [1,1,1,1,0,0,0,0,0,1,0]

Once again, the Figures 1.5 and 1.6 demonstrate the visualisation of the class voting but this time for **Y = 0**. These results meet the objectives set in **G3** and produced a visual representation of the clause summation and class voting as expected. In terms of drawing clear results from this section, the visual tool was a success. Although this implementation is simple in design, using a parsing tool to read the clauses, the idea is novel and is the first version created, hence it could be taken further with larger datasets to be used by students when first learning about the TM. This is exactly as discussed in the introduction. In some way, the simplistic approach to building such a tool is almost complimentary to the indented use, such that the theory behind the script i.e., parsing clauses, summation etc. is exactly the best way to incrementally learn the dynamics of the TM voting methods. On a different level, higher-level versions of this tool (following the same sort of concept) could be used for example to explain the reasons for a high-stake decision made by a model and would verify validity of that decision.

2. Visualisation of TA flips

The next target for the project was to plot the average number of flips within the TM, observing the difference in plots after varying the TM parameters of learning sensitivity, ‘s’, and learning thresholding, ‘T’. Using a C/C++ TM toolkit [17], the user can choose parameters of the machine and specify which data they wish to extract, saving it to a CSV file to then be used in external visualisation. In this case, a Python script was created for such visualisation. The following figures show the plots for such data, the first figure showing the variation of the learning rate ‘s’ and the second showing the variation of the threshold value ‘T’. These parameters control the level of stochasticity within the feedback provided to the TA states, where ‘s’ dictates the probability of feedback versus inaction, whilst ‘T’ decides the likelihood of feedback over time, by controlling the target number of clauses necessary to activate. In doing so, this allows for model fine tuning and mitigates the risk of overfitting.

The other parameters for training the TM with the 10-class MNIST dataset are as follows:

100 clauses per class,
100 include
100 exclude
} states per TA

The measurement for data logging has a step size of 2000 input vectors, with the TM training 100 steps meaning that there are 33 log entries per epoch in the case of training the MNIST, and therefore 100 inputs is equal to training for 3 epochs.

a) Varying learning rate ‘s’

In the first half of this experiment, ‘s’ was varied, ranging from s=2.0 to s=10.0, at increments of 2, whilst ‘T’ was set to a constant value 17.5. It was decided that this would give good distribution of sensitivity values, whilst also maintaining a good level of accuracy (as shown in both plot legends). It must be noted that the graphs use a logarithmic scale on the y-axis because this provides more resolute plot of the data, which was not the case in the original, using non-logarithmic axis, due to the much larger starting average number of TA flips.

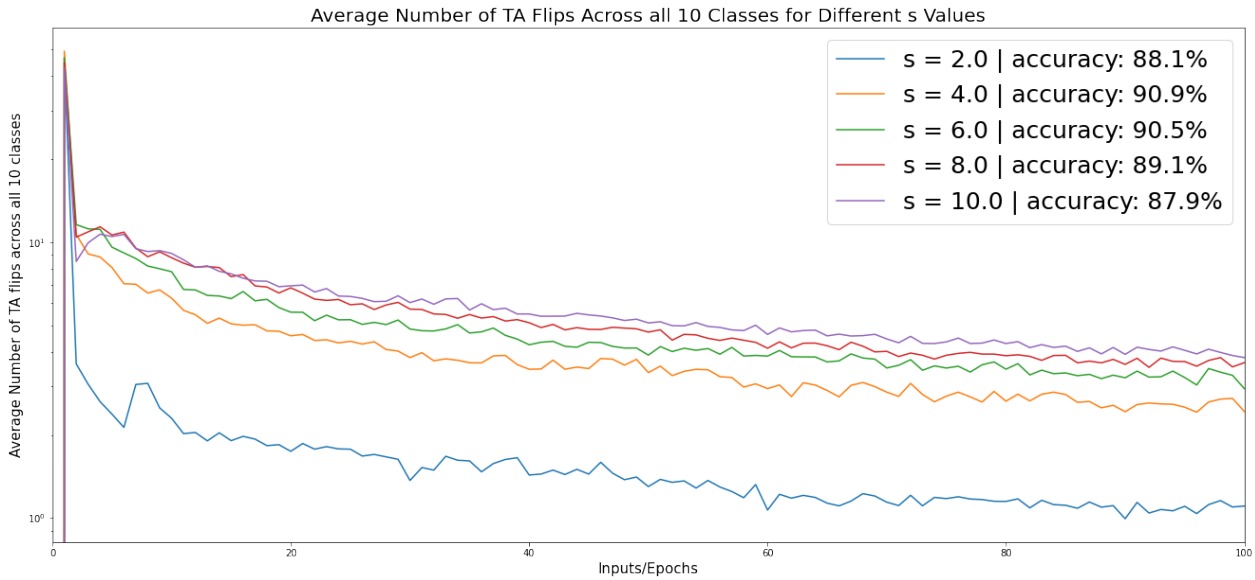

Fig 1.7 The average number of TA flips across all 10 classes for different values of ‘s’

In the above Fig 1.7, for all values of ‘s’, the starting average number of TA flips (referred to as average number of flips, ANOF, for simplicity) is very high due to the TAs initial random assignment to the middle states (state boundaries). As the TM trains however, each TA becomes more confident in its own state and so the flickering between include and exclude occurs less and less. Noticeably, with values of s=4.0, shown by the blue line, the ANOF is much lower than the ANOF at other values of ‘s’. The ANOF starts high, as is the trend, but immediately drops lower than any other lines as the training starts. This can be explained since the effects of changing the ‘s’ values follow these probabilistic feedback equations:

$$Include\ is\ rewarded\ and\ exclude\ penalized\ with\ probability\ \frac{s-1}{s}$$

$$Include\ is\ penalized\ and\ exclude\ rewarded\ with\ probability\ \frac{1}{s}$$

As such, with lower ‘s’ values, we can expect to see TA bias towards favouring the exclude state, which is exactly what is presented in the above plot. As you increase ‘s’, the number of included states also increase and so the TAs tend to have a greater ANOF in order to “crossover” into the include state.

Another interesting observation can be made by focussing on the section of Figure 1.7 after 20 inputs. Although it can be seen for all levels of ‘s’, the ANOF starts to somewhat smoothen out, looking specifically at s=2.0, the range between the local maximum and local minimum is comparatively greater than that of any other line. This means, at this ‘s’ value, the rate of change of ANOF is the

greatest, thus the TAs must be the most unstable. A trend can be observed showing as the 's' value increases, the rate of change of the ANOF decreases. It can therefore be said that the 's' value is inversely proportional to the rate of change of the average number of TA flips. To confirm this observation, further analysis could be performed such as the application of an exponential weighted moving average (EWMA). This would produce a standard deviation across the curved line to determine the rate of change precisely. In terms of hardware-implementation, a lower number of TA flips is beneficial because this reduces the amount of switching, hence requiring less power.

Finally, the legend displays the accuracy of each respective learning rate value, showing a peak accuracy of 90.9% at s=4.0, after which the accuracy begins to fall as 's' was increased further. An accuracy of this level is considered good, especially after only training for 3 epochs with such a large dataset.

b) Varying learning threshold 'T'

The second half of the experiment involved increasing the learning rate 'T' value from T=12.5 to T=22.5 at increments of 2.5, whilst keeping 's' at a constant value of 3.9.

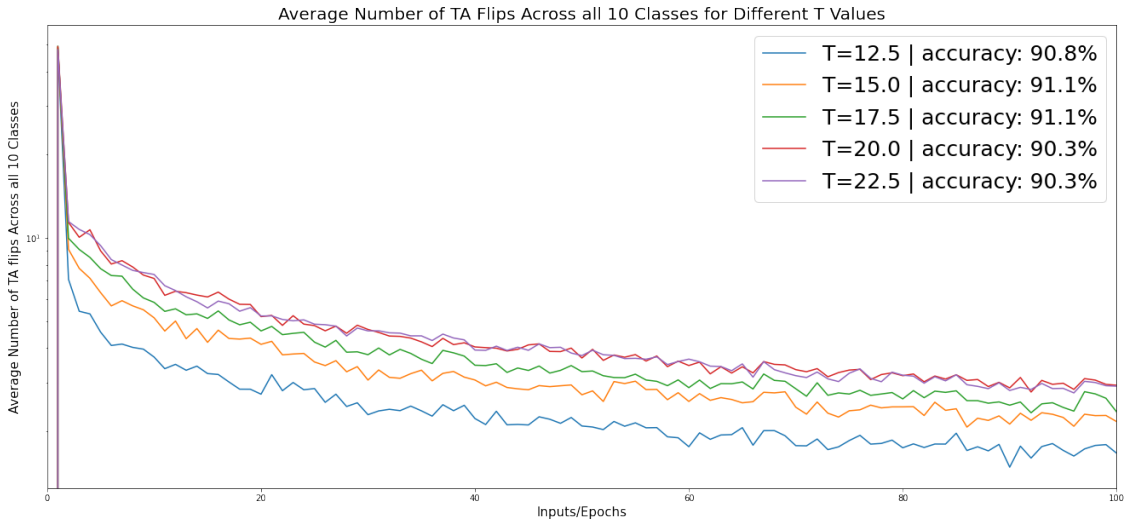

Fig 1.8 The average number of TA flips across all 10 classes for different values of 's'

A similar trend seen in the Figure 1.7 can also be seen in Figure 1.8. The lower values of the parameter 'T' produce a lower ANOF than the higher values. Considering the accuracy in this experiment, the average accuracy across all values of 'T' is 90.72%. Unlike in the case of varying 's' parameter, the accuracy remains very similar-throughout this experiment which may be consequential of the fact the TM was only trained for 3 epochs. When comparing the two plots, Figure 1.8 clearly shows a much lower ANOF at the same relative number of inputs/epochs. For example, looking at the ANOF at 20 inputs/epoch for both figures, all the 'T' value ANOF are significantly lower than the ANOF for the differing 's' values. This is because the likelihood of feedback is decreased as 'T' increases meaning there are fewer changes to the TA states, hence fewer flips between include and exclude.

In terms of these results, this type of visualisation can help develop an understanding of occurrences such as the ones above. Linking back to Section I, this would be an example of how visualisation can be used to study different ML algorithms and the effects of varying the hyper-parameters. Both figures in the above sections successfully met all the objectives associated with **G4**.

3. Locally Stochastic Clause Dropping

The final more ambitious goal for this project was to propose an original way in which the TM architecture could be improved. The following results were taken from training the Noisy XOR dataset where the TM was written in C. An idea was proposed within the later stages of the project (due to a greater understanding of the TM) where the clause outputs for each clause per class could be gathered after each epoch. During training, whilst still undergoing feedback, the 'vote winning' classes are identified. From there, the least useful clauses within that class are found. These relative superfluous clauses could then be grouped separately, with the intention of randomly removing one of the clauses within that group. Furthermore, a new parameter could then be introduced that decides what percentage of the grouped 'less useful' clauses are removed, for example 5%. This takes inspiration from the aforementioned idea of stochastic clause dropping proposed by Granmo [13], alternatively in this case, the dropping of clauses would be local stochastic removal as opposed to absolute stochastic removal. One point to note would be that this idea would be much better implemented with a larger dataset such as the MNIST using many more clauses than just 10 per class. A greater number of clauses would allow for more clauses to be removed, over a greater number of epochs, whilst still maintaining the integrity of accuracy. A further analysis of this concept would be to vary the 's' values as the more 'useless' clauses are determined and then observe the effects.

To achieve this concept of redundant clause identification, a visualisation method can be implemented where each of the clause values are summed over the training process of 100 epochs and plotted as such below. This idea proved to be simple but effective in finding the highest frequency of clauses producing 0, visibly presenting the contrast between more useful clauses versus that of the less utilised clauses.

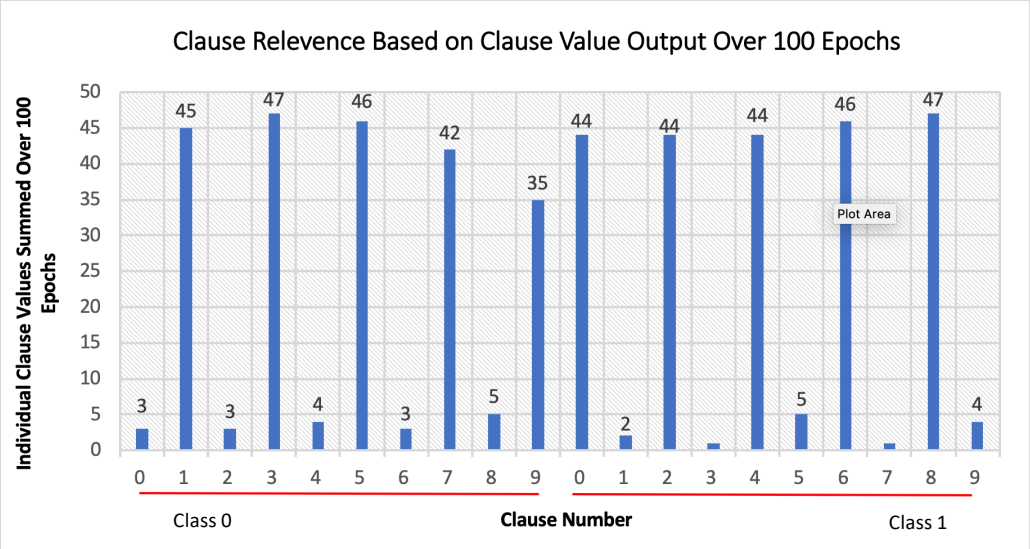

Fig 1.9 A chart to show the calculated clause relevance over 100 epochs

Fig 1.9 shows the summation of all clause values. The least used clauses within the winning classes are acknowledged as:

Class 0: Clause 9
Class 1: Clause 0 or 2 or 4

Taking these very simple results for example, it could be said that removing the above identified clauses would still result in accurate predictions, whilst also having positive effects on training-time, interpretability and robustness of the TM. Of-course, these results are only scratching the surface of

what would need to be done in order to implement such an idea with a larger dataset. With that being said, these results do in fact demonstrate the basic concept of how to approach said task. This is yet another example of how visualisation can lead to a greater understanding of the internal dynamics of the TM, again linking back to both studying new algorithms and also improving said algorithms. Due of the ideas discussed here, it could be argued that **G5** was a moderate success with some good ideas presented.

VI. Conclusion

1. Key Highlights

This work explores the different possibilities associated with visualising ML and more specifically the TM implementation. Section I and II provide insight into the importance of interpretability and the power of visualisation, specifically demonstrating how they can be useful when studying ML theory. Another demonstrated use for visualisation is analysing low-complexity architectures for performance enhancement and verifying the veracity of predictions to improve prediction dependability. Section III develops background theory of the TM, whilst section IV presents examples of visualisation results achieved in this project.

Notably within Section IV, a visualisation tool was implemented which enabled the discovery of an interesting relationship between learning rate ‘s’ and number of flips, demonstrating the importance of these tools within ML in general. Not only does the visual tool work as intended but it also has provided a useful insight into the inner workings of the TM algorithm. Furthermore, an explanation for this observation has been proposed based on the progressive understanding of the TM developed throughout this project. Ideas of the benefits involved with creating visual tools were always related back to the key ideas discussed in the Section I Introduction.

2. State of Objectives Fulfilled

The aims and objectives have been fulfilled, with the slight exception of **G5**. In addition, **G6** and **G7** have both been achieved in that these skills have been largely developed throughout this project considering they were notably undeveloped to begin with.

3. Self-Critique

A critical analysis of this project would conclude upon a relative success. Methods of improvement include the use of datasets with greater complexity. This would provide more data to visualise hence allowing for greater, in-depth visualisation.

4. Future Works

Finally, concluding the paper, a novel idea was proposed with basic results and discussion regarding the collection of the data required to implement such an idea. There is definitely a greater amount of work to be done on this concept, for example, once the locally stochastic clause dropping is implemented, run tests on the model using different values of learning rate ‘s’ and observe the difference in number of clause redundancies.

VII. References

[1] A. Chatzimparmpas, R. M. Martins, I. Jusufi, and A. Kerren, 'A survey of surveys on the use of visualization for interpreting machine learning models', *Inf. Vis.*, vol. 19, no. 3, pp. 207–233, Jul. 2020, doi: 10.1177/1473871620904671.

[2] M. T. Ribeiro, S. Singh, and C. Guestrin, '"Why Should I Trust You?": Explaining the Predictions of Any Classifier'. arXiv, Aug. 09, 2016. Accessed: May 17, 2022. [Online]. Available: http://arxiv.org/abs/1602.04938

[3] O.-C. Granmo, 'An Introduction to Tsetlin Machines'. 2021. [Online]. Available: http://tsetlinmachine.org/wp-content/uploads/2021/09/Tsetlin_Machine_Book_Chapter_1-4.pdf

[4] O.-C. Granmo, 'The Tsetlin Machine -- A Game Theoretic Bandit Driven Approach to Optimal Pattern Recognition with Propositional Logic', *ArXiv180401508 Cs*, Jan. 2021, Accessed: May 06, 2022. [Online]. Available: http://arxiv.org/abs/1804.01508

[5] D. Fumo, 'A Gentle Introduction To Neural Networks Series', vol. Part 1, Aug. 2017, [Online]. Available: https://towardsdatascience.com/a-gentle-introduction-to-neural-networks-series-part-1-2b90b87795bc

[6] J. Lei, A. Wheeldon, R. Shafik, A. Yakovlev, and O.-C. Granmo, 'From Arithmetic to Logic based AI: A Comparative Analysis of Neural Networks and Tsetlin Machine', in *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, Glasgow, UK, Nov. 2020, pp. 1–4. doi: 10.1109/ICECS49266.2020.9294877.

[7] C. Molnar, *Interpretable Machine Learning: A Guide for Making Black Box Models Interpretable*. 2022.

[8] A. Mordvintsev, 'Inceptionism: Going Deeper into Neural Networks'. 2015. [Online]. Available: https://research.google/pubs/pub45507/

[9] Z. C. Lipton, 'In machine learning, the concept of interpretability is both important and slippery.', *Mach. Learn.*, p. 28.

[10] R. Saha, O. Granmo, and M. Goodwin, 'Using Tsetlin Machine to discover interpretable rules in NATURAL LANGUAGE PROCESSING applications', *Expert Syst.*, Nov. 2021, doi: 10.1111/exsy.12873.

[11] A. Phoulady, O.-C. Granmo, S. R. Gorji, and H. A. Phoulady, 'The Weighted Tsetlin Machine: Compressed Representations with Weighted Clauses'. arXiv, Jan. 14, 2020. Accessed: May 18, 2022. [Online]. Available: http://arxiv.org/abs/1911.12607

[12] K. D. Abeyrathna, O.-C. Granmo, and M. Goodwin, 'Extending the Tsetlin Machine With Integer-Weighted Clauses for Increased Interpretability', *ArXiv200505131 Cs*, May 2020, Accessed: May 12, 2022. [Online]. Available: http://arxiv.org/abs/2005.05131

[13] J. Sharma, R. Yadav, O.-C. Granmo, and L. Jiao, 'Drop Clause: Enhancing Performance, Interpretability and Robustness of the Tsetlin Machine', *ArXiv210514506 Cs*, Jan. 2022, Accessed: May 12, 2022. [Online]. Available: http://arxiv.org/abs/2105.14506

[14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, 'Dropout: A Simple Way to Prevent Neural Networks from Overfitting', p. 30.

[15] O.-C. Granmo, *Interpretability Demo*. 2022.

[16] O.-C. Granmo, *Tsetlin Machine Interpretability Demo*. 2021. [Online]. Available: https://github.com/cair/pyTsetlinMachine#interpretability-demo

[17] A. Rafiev, *Tsetlin Tutorial*. 2022. [Online]. Available: https://github.com/ashurrafiev/TsetlinTutorial