1. An Introduction to Logical Interpretable Learning with the Tsetlin Machine

1.1 The Concept of Logical Interpretable Learning

Consider a well-known logical exclusive OR (XOR) operator defined with a truth table in Figure 1. Each row in the table represents values of input variables x1, x2 and a corresponding output variable y. The inputs of each row can also be expressed as a conjunctive clause of literals (a literal is either an input variable or its negations): ^x1 &^x2, ^x1 & x2, x1 &^x2, and x1 & x2. Note, that clauses ^x1 & x2 and x1 &^x2 correspond to *1 (true) output values*, while clauses ^x1 &^x2 and x1 & x2 correspond to *0 (false) output value.* We say that clauses ^x1 & x2 and x1 &^x2 have *positive polarity (positive clauses)*. Similarly, clauses ^x1 &^x2 and x1 & x2 have *negative polarity (negative clauses)*. Thus, true output 1 can be expressed as a disjunction of conjunctive clauses with positive polarities, as shown in Figure 1. Similarly, a disjunction of conjunctive clauses with negative polarities expresses false output 0.

	x1 x2 y	**clause**	**polarity**	Disjunction of clauses with positive polarities to express output y = 1:
XOR	0 0 0	^x1 &^x2	negative (-)	**(x1 &^x2) or (x2 &^x1)**
Truth Table:	0 1 1	^x1 & x2	positive (+)	
	1 0 1	x1 & ^x2	positive (+)	Disjunction of clauses with negative polarities to express output y = 0:
	1 1 0	x1 & x2	negative (-)	**(^x1 &^x2) or (x2 &x1)**

Figure 1: Exclusive OR operator

The logical interpretable learning aims at discovering (learning) *positive* and *negative* clauses from an arbitrary truth table. We assume that rows of the truth table are produced *one by one* from an *incoming data stream*. As a new row arrives, the learner tries to guess the positive and negative clauses by deciding on whether to *include* certain literals in a clause. The incoming data stream can be *noisy*: we will consider the impact of the noise on the learning process and ways to mitigate it in the next section. For simplicity, in this section we will illustrate the process of logical interpretable learning assuming that the incoming data stream is noise-free.

Intuitively, the more incoming rows the leaner explores the more accurate its guesses should be. The process of learning a clause should *accumulate evidences* from the rows of the incoming data stream in order to decide whether *to include* a literal in a conjunctive clause, or *to exclude* it from the clause. We will explain this approach with a simple evidence accumulation method, illustrated in Figure 2. This method, which we call Naïve Counting, suggests including a literal in a *positive clause* by *incrementing* the literal inclusion counter *if the value of the literal is 1 and the output is also 1* (i.e., the literal "agrees" with a positive output). *If the literal value is 1 and the output is 0*, then the literal "agrees" with the negative output and our Naïve Counting method suggests including such a literal in a *negative clause* (by *incrementing* the literal counter for the negative clause). Figure 2 shows how the inclusion counters are updated for each literal in a clause of positive and negative polarity. At the end of the data stream, for n literals, the final conjunctive clause will include n/2 literals with the largest inclusion counter value (top n/2 literals).

The rules for updating inclusion counters are reflected in a *feedback greed*, which suggests an update for each literal counter (FEEDBACK column) based on the *clause polarity*, as well as on *output* and *literal* values in the incoming data stream. Missing FEEDBACK value corresponds to 0 (no update). In this example the incoming data stream is free of noise, and we managed to learn a correct positive clause x1&^x2, and a correct negative clause ^x1&^x2.

Learning Logical Expressions from Data: Naïve Counting

	x1	x2	y
	0	0	0
XOR:	0	1	1
	1	0	1
	1	1	0

(x1 &^x2) or (x2 &^x1)

Literal inclusion counters for +/- polatities:

Incoming Data Stream			**positive (+)**				**negative (-)**			
x1	x2	y (**output**)	x1	x2	^x1	^x2	x1	x2	^x1	^x2
			0	0	0	0	0	0	0	0
0	0	0					0	0	1	1
0	1	1	0	1	1	0				
0	0	0					0	0	2	2
0	1	1	0	2	2	0				
1	0	1	1	2	2	1				
1	1	0					1	1	2	2
0	0	0					1	1	**3**	**3**
1	0	1	2	2	2	2				
1	0	1	**3**	2	2	**3**				

x1 &^x2 **^x1 &^x2**

include top 2 literals (for n literals, top n/2 literals)

Feedback Grid:

polarity	***output***	***literal***	***FEEDBACK***
+	0	0	
+	0	1	
+	1	0	
+	1	1	+1
-	0	0	
-	0	1	+1
-	1	0	
-	1	1	

if output=1 & literal=1, then positive counter increment
if output=0 & lieralt=1, then negative counter increment

Figure 2: Learning based on Naïve Counting with noise-free data

1.2 Adjusting feedback to mitigate the noise

The incoming data stream can be noisy. We consider two kinds of noise, as illustrated in Figure 3: (1) *0-noise, or false noise*, where correct output 0 is replaced by 1, and (2) *1-noise, or true noise*, where correct output 1 is replaced by 0. The noisy data stream in Figure 3 includes only two correct rows (black output). The first row is a 0-noise, while the second and the last rows are 1-noises.

(x1 &^x2) or (x2 &^x1)

XOR			Noisy XOR			
x1	x2	y	x1	x2	y	
0	0	0	0	0	1	0->1: **0-noise (false noise)**
0	1	1	0	1	0	1->0: 1-**noise (true noise)**
1	0	1	1	0	0	1->0: **1-noise (true noise**
1	1	0	1	1	1	0->1: **0-noise (false noise)**

Noisy Data Stream		
x1	x2	y (**output**)
0	0	1
0	1	0
0	0	0
0	1	1
1	0	0

Figure 3: Examples of a noise and a noisy data stream

Next we consider how a noise impacts the learning. Figure 4 illustrates the Naïve Counting learning from a noisy data stream. We observe that both positive and negative conjunctive clauses were learned incorrectly. Moreover, in the negative clause both x1 and ^x1 have the second largest inclusion counter value 2. We could enforce including only n/2 literals by randomly selecting either x1 or ^x1 to include in the resulting clause. However, it gives us a chance to learn an incorrect negative clause x1& ^x2.

Learning Logical Expressions from NOISY Data: Naïve Counting

(x1 &^x2) or (x2 &^x1)

XOR	Noisy XOR	
0 0 0	0 0 1	0->1: **0-noise (false noise)**
0 1 1	0 1 0	1->0: 1-**noise (true noise)**
1 0 1	1 0 0	1->0: **1-noise (true noise**
1 1 0	1 1 1	0->1: **0-noise (false noise)**

Literal inclusion counters for +/- polatities:

Noisy Data Stream			**positive (+)**				**negative (-)**			
x1	x2	y **(output)**	x1	x2	^x1	^x2	x1	x2	^x1	^x2
			0	0	0	0	0	0	0	0
0	0	1	0	0	1	1				
0	1	0					0	1	1	0
0	0	0					0	1	2	1
0	1	1	0	1	2	1				
1	0	0					1	1	2	2
1	1	1	1	2	2	1				
0	0	1	1	2	3	2				
1	0	1	2	2	**3**	**3**				
1	0	0					**2**	1	**2**	**3**

^x1 &^x2   **x1 &^x1 &^x2 ?**

include top 2 literals (for n literals, top n/2 literals)

Feedback Grid:

polarity	***output***	***literal***	***FEEDBACK***
+	0	0	
+	0	1	
+	1	0	
+	1	1	+1
-	0	0	
-	0	1	+1
-	1	0	
-	1	1	

if output=1 & literal=1, then positive counter increment
if output=0 & lieralt=1, then negative counter increment

Figure 4: Impact of noise

In order to reduce the noise impact we will extend the feedback, as illustrated in Figure 5: in addition to the *clause polarity*, *output* and *literal* values we will also consider a *current value of the clause*. At each learning step we will estimate the clause value using the n/2 literals with the largest current inclusion counters. Similar to the basic feedback, the extended feedback suggests incrementing the inclusion literal counters for a positive clause if the value of the literal is 1 and the output is also 1, and for a negative clause, if the literal value is 1 and the output is 0. However, the counters will be incremented only if the current clause value is estimated to 1 (i.e., the literals also “agree” with the current clause estimates). Figure 5 shows the extended feedback grid and illustrates its performance on the noise-fee incoming data stream. We managed to learn a correct positive clause ^x1& x2, and a correct negative clause ^x1&^x2, although the results are slightly different from what we obtained using a basic feedback in Figure 3.

XOR — Learning Logical Expressions using Naïve Counting with Extended Feedback (using clause vlue evaluated with top n/2 literals)

XOR	
0 0 0	if output=1 & literal=1 & clause = 1 then positive counter increment
0 1 1	if output=0 & literal=1 & clause = 1 then negative counter increment
1 0 1	(emty clause is evaluated to 1)
1 1 0	**(x1 &^x2) or (x2 &^x1)**

Incoming Data Stream

x1	x2	y **(output)**	**positive(+)** x1	x2	^x1	^x2	***clause***	**negative(-)** x1	x2	^x1	^x2	***clause***
			0	0	0	0		0	0	0	0	
0	0	0						0	0	1	1	1
0	1	1	0	1	1	0	1					
0	0	0						0	0	2	2	1
0	1	1	0	2	2	0	1					
1	0	1	1	2	2	1	1					
1	1	0						0	0	2	2	0
0	0	0						0	0	**3**	**3**	1
1	0	1	1	2	2	1	0					
1	0	1	1	**2**	**2**	1	0					
			^x1 & x2					**^x1 &^x2**				

top 2 literals (for n literals, top n/2 literals)

Extended Feedback Grid:

polarity	***output***	***clause***	***literal***	***FEEDBACK***
+	0	0	0	
+	0	0	1	
+	0	1	0	
+	0	1	1	
+	1	0	0	
+	1	0	1	
+	1	1	0	
+	1	1	1	+1
-	0	0	0	
-	0	0	1	
-	0	1	0	
-	0	1	1	+1
-	1	0	0	
-	1	0	1	
-	1	1	0	
-	1	1	1	

Figure 5: Learning with Extended Feedback

Consider now how a noise impacts the learning with the extended feedback. This is illustrated in Figure 6. We observe that the positive conjunctive clause was learned correctly, which is encouraging. However, learning the negative clause resulted in the same inclusion counters for *all literals*, which is confusing. To reduce the noise impact some of those literals should be *excluded* from the clause, i.e., we need to *decrement some literal inclusion counters*. Next, we will adjust the feedback to enable such exclusion.

XOR — Learning from Noisy Data using Extended Feedback Grid

XOR	
0 0 0	if output=1 & literal=1 & clause = 1 then positive counter increment
0 1 1	if output=0 & literal=1 & clause = 1 then negative counter increment
1 0 1	
1 1 0	**(x1 &^x2) or (x2 &^x1)**

Noisy Data Stream

x1	x2	y **(output)**	**positive(+)** x1	x2	^x1	^x2	***clause***	**negative(-)** x1	x2	^x1	^x2	***clause***
			0	0	0	0		0	0	0	0	
0	0	1	0	0	1	1	1					
0	1	0						0	1	1	0	1
0	0	0						0	1	1	0	0
0	1	1	0	1	2	1	1					
1	0	0						0	1	1	0	0
1	1	1	1	2	2	1	1					
0	0	1	1	2	2	1	0					
1	0	1	1	**2**	**2**	1	0					
1	0	0					0	1	1	1	1	1
			^x1 & x2					**x1 & x2 &^x1 &^x2 ?**				

top 2 literals (for n literals, top n/2 literals)

Extended Feedback Grid:

polarity	***output***	***clause***	***literal***	***FEEDBACK***
+	0	0	0	
+	0	0	1	
+	0	1	0	
+	0	1	1	
+	1	0	0	
+	1	0	1	
+	1	1	0	
+	1	1	1	+1
-	0	0	0	
-	0	0	1	
-	0	1	0	
-	0	1	1	+1
-	1	0	0	
-	1	0	1	
-	1	1	0	
-	1	1	1	

Figure 6: Noise impact on learning with Extended Feedback

Figure 7 explains the feedback with the inclusion counter *decrement*, which is a *suggestion to exclude* the literal from a clause. Namely, we will decrement the literal inclusion counter for a negative clause *if the output is 0 and the literal is 0* (informally, when the literal of a clause with a negative polarity "disagrees" with a negative output). As we observe, the counter decrement helps to mitigate the noise impact and we managed to learn a correct positive clause ^x1& x2, and a correct negative clause ^x1&^x2.

XOR	Learning from Noisy Data using Extended Feedback with counter decrement
0 0 0	if output=1 & literal=1 & clause = 1 then positive counter increment
0 1 1	if output=0 & literal=1 & clause = 1 then negative counter increment
1 0 1	if output=0 & literal=0 & then negative counter decrement
1 1 0	**(x1 &^x2) or (x2 &^x1)**

Noisy Data Stream

x1	x2	y (output)	**positive(+)** x1	x2	^x1	^x2	***clause***	**negative(+)** x1	x2	^x1	^x2	***clause***
			0	0	0	0		0	0	0	0	
0	0	1	0	0	1	1	1					
0	1	0						-1	1	1	-1	1
0	0	0						-2	0	1	-1	0
0	1	1	0	1	2	1	1					
1	0	0						-2	-1	0	-1	0
1	1	1	1	2	2	1	1					
0	0	1	1	2	2	1	0					
1	0	1	1	**2**	**2**	1	0					
1	0	0					0	-2	-2	-1	-1	0

^x1 & x2 **^x1 &^x2**

top 2 literals (for n literals, top n/2 literals)

Extended Feedback Grid:

polarity	***output***	***clause***	***literal***	***FEEDBACK***
+	0	0	0	
+	0	0	1	
+	0	1	0	
+	0	1	1	
+	1	0	0	
+	1	0	1	
+	1	1	0	
+	1	1	1	+1
-	0	0	0	-1
-	0	0	1	
-	0	1	0	-1
-	0	1	1	+1
-	1	0	0	
-	1	0	1	
-	1	1	0	
-	1	1	1	

Figure 7: Noise mitigation with literal inclusion counter decrement

To sum up, different feedback strategies can mitigate the impact of the noise in the process of the logical interpretable learning. We considered several examples to illustrate this point. *Later we explore more refined feedback strategies used in advanced Tsetlin Machine learning method, that dramatically improves performance of the logical interpretable learning.*

1.3 Learning multiple clauses

The Naïve Counting approach that we introduced in the previous sections learns *exactly one* positive and one negative conjunctive clause. Moreover, since the Naïve Counting updates all the literal inclusion counters deterministically, the same incoming data stream will result in the same pair of positive and negative conjunctive clauses for each run of the Naïve Counting. Meanwhile, as we observed in Figure 1, a comprehensive characterization of the logical output requires a *disjunction of multiple conjunctive clauses*. A disjunction of clauses with positive polarities (x1 &^x2) or (x2 &^x1) would express output y = 1; a disjunction of clauses with negative polarities (^x1 &^x2) or (x2 &x1) would express output y = 0.

In order to make sure that the learning has a chance to explore multiple clauses of positive and negative polarities we extend the Naïve Counting with a *probabilistic counter updates*, as illustrated in Figure 8. To simplify the explanation, we will use Naïve Counting with the basic feedback grid, as shown in Figure 2. If the literal counter update condition is met (e.g., a value of the literal in a positive clause is 1 and the output is also 1) the update will be performed with a probability *pcu* (i.e., if a randomly generated number *rand* is less then *pcu*). Figure 8 shows that we managed to learn correct positive and negative clauses. However, the inclusion counter values are *different* from the values in Figure 2, since the updates were performed probabilistically. In contrast to the basic non-probabilistic Naïve Counting, each run of Naïve Counting with probabilistic updates may result in *different values of the literal inclusion counters even for the same incoming data stream*. We will use this property of the probabilistic Naïve Counting to learn multiple positive and negative conjunctive clauses, as illustrated in Figure 9.

XOR — Learning Logical Expressions with Probabilistic Update

0 0 0 — With update probability pcu:

0 1 1 — if rand <= pcu & output=1 & literal=1, then positive counter increment

1 0 1 — if rand <= pcu & output=0 & lieralt=1, then negative counter increment

1 1 0

(x1 &^x2) or (x2 &^x1)

Incoming Data Stream			**positive(+)**				**negative(-)**				pcu = 0.5
x1	x2	y **(output)**	x1	x2	^x1	^x2	x1	x2	^x1	^x2	
			0	0	0	0	0	0	0	0	rand:
0	0	0					0	0	1	1	0.3 (update)
0	1	1	0	0	0	0					0.6 (no update)
0	0	0					0	0	1	1	0.7 (no update)
0	1	1	0	1	1	0					0.4 (update)
1	0	1	0	1	1	0					0.8 (no update)
1	1	0					1	1	1	1	0.2 (update)
0	0	0					1	1	**2**	**2**	0.3 (update)
1	0	1	1	1	1	1					0.1 (update)
1	0	1	**2**	1	1	**2**					0.3 (update)

x1 &^x2 — **^x1 &^x2**

top 2 literals (for n literals, top n/2 literals)

Figure 8: Naïve Counting with probabilistic updates

The main idea of the proposed approach is to *allocate separate literal inclusion counters for several positive and negative clauses*. In Figure 9 we allocate literal inclusion counters for two positive clauses c1 and c3, and for two negative clauses c2 and c4[1]. Since the counters will be updated probabilistically, we expect to have different literal inclusion counter values for each clause, and we are likely to learn *four different clauses*. This is really the case shown in Figure 9. Moreover, the disjunction of learned clauses c1, c3 with positive polarities (x2 &^x1) or (x1 &^x2) *correctly expresses the output y = 1*, while the disjunction of learned clauses c2, c4 with negative polarities (x2 &x1) or (^x1 &^x2) *correctly expresses the output y = 0*.

[1] Here and below we will use the following clause indexing agreement: positive clauses will have odd indices, while negative clauses will have even indices.

XOR	
0 0 0	Learning Logical Expressions with Probabilistic Update and with Multiple Clauses
0 1 1	With update probability pcu:
1 0 1	For each clause ci:
1 1 0	if rand <= pcu & output=1 & literal=1, then positive counter increment
	if rand<=pcu & output=0 & literal=1, then negative counter increment

(x1 &^x2) or (x2 &^x1)

Incoming Data Stream				**positive(+)**					**negative(+)**				pcu = 0.5
x1	x2	y **(output)**		x1	x2	^x1	^x2		x1	x2	^x1	^x2	
			c1:	0	0	0	0	c2:	0	0	0	0	rand:
			c3:	0	0	0	0	c4:	0	0	0	0	
0	0	0							0	0	0	0	0.6 (no update)
									0	0	1	1	0.3 (update)
0	1	1		0	1	1	0						0.4 (update)
				0	0	0	0						0.7 (no update)
0	0	0							0	0	0	0	0.8 (no update)
									0	0	2	2	0.2 (update)
0	1	1		0	2	2	0						0.3 (update)
				0	0	0	0						0.9 (no update)
1	0	1		0	3	3	0						0.3 (update)
				1	0	0	1						0.1 (update)
1	1	0							1	1	0	0	0.2 (update)
									0	0	2	2	0.8 (no update)
0	0	0						c2:	**1**	**1**	0	0	0.7 (no update)
								c4:	0	0	**3**	**3**	0.3 (update)
1	0	1		0	3	3	0						0.7 (no update)
				2	0	0	2						0.2 (update)
1	0	1	c1:	0	3	3	0						0.6 (no update)
			c3:	**2**	0	0	**2**						0.8 (no update)

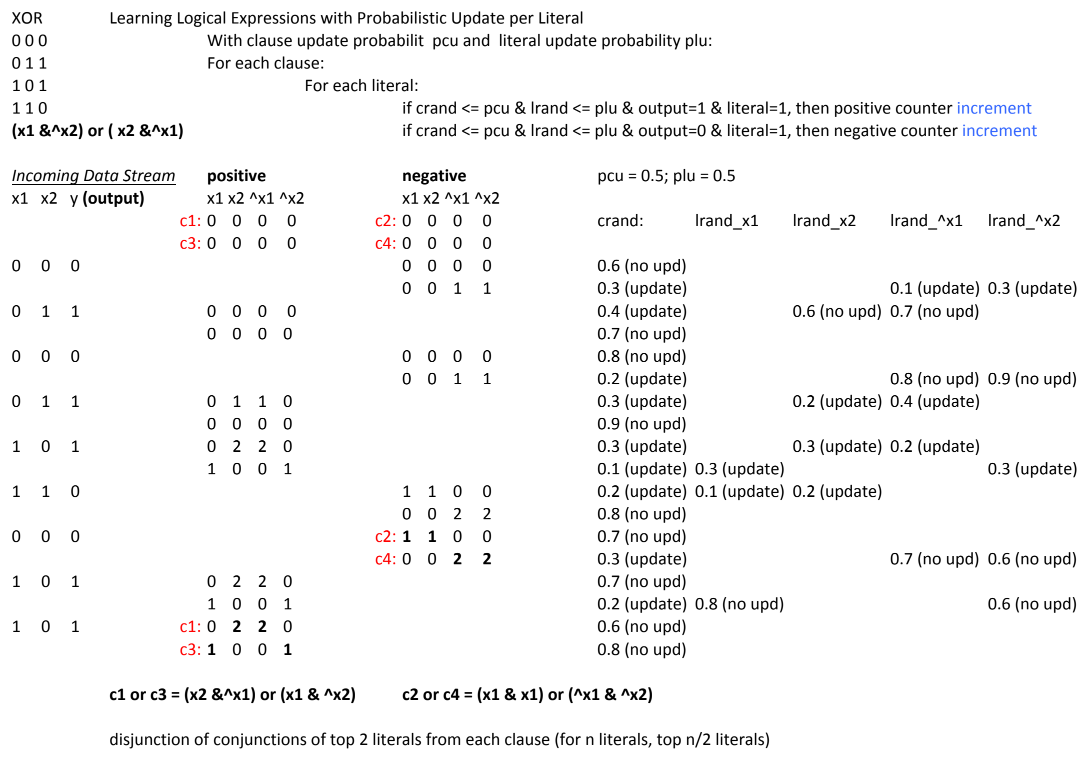
c1 or c3 = (x2 &^x1) or (x1 & ^x2) **c2 or c4 = (x1 & x1) or (^x1 & ^x2)**

disjunction of conjunctions of top 2 literals from each clause (for n literals, top n/2 literals)

Figure 9: Using probabilistic updates to generate multiple clauses

The above probabilistic method applies updates to all literal inclusion counters for a clause with a probability *pcu*. We can further extend the probabilistic framework associating an update probability *plu* with each literal inclusion counter, as illustrated in Figure 10. This solution is more flexible and would allow the learner to explore a wider range of potential clauses. As we observe in Figure 10, we managed to learn correct disjunctive clauses, although the counter values are different, as expected.

You may notice that the learning outcome could change as the probabilities change. *One of the major goals of a sustainable logical interpretable learning is to devise a probabilistic framework, which would increase the chances of learning a correct and comprehensive set of clauses characterizing the incoming data stream.* Later we will explain how the Tsetlin Machine achieves this goal.



XOR

0 0 0

0 1 1

1 0 1

1 1 0

(x1 &^x2) or (x2 &^x1)

Learning Logical Expressions with Probabilistic Update per Literal

```
With clause update probabilit  pcu and  literal update probability plu:
For each clause:
    For each literal:
        if crand <= pcu & lrand <= plu & output=1 & literal=1, then positive counter increment
        if crand <= pcu & lrand <= plu & output=0 & literal=1, then negative counter increment
```

pcu = 0.5; plu = 0.5

Incoming Data Stream			**positive**		**negative**					
x1	x2	y **(output)**		x1 x2 ^x1 ^x2		x1 x2 ^x1 ^x2	crand:	lrand_x1	lrand_x2	lrand_^x1	lrand_^x2
			c1:	0 0 0 0	c2:	0 0 0 0					
			c3:	0 0 0 0	c4:	0 0 0 0					
0	0	0				0 0 0 0	0.6 (no upd)				
						0 0 1 1	0.3 (update)			0.1 (update)	0.3 (update)
0	1	1		0 0 0 0			0.4 (update)		0.6 (no upd)	0.7 (no upd)	
				0 0 0 0			0.7 (no upd)				
0	0	0				0 0 0 0	0.8 (no upd)				
						0 0 1 1	0.2 (update)			0.8 (no upd)	0.9 (no upd)
0	1	1		0 1 1 0			0.3 (update)		0.2 (update)	0.4 (update)	
				0 0 0 0			0.9 (no upd)				
1	0	1		0 2 2 0			0.3 (update)		0.3 (update)	0.2 (update)	
				1 0 0 1			0.1 (update)	0.3 (update)			0.3 (update)
1	1	0				1 1 0 0	0.2 (update)	0.1 (update)	0.2 (update)		
						0 0 2 2	0.8 (no upd)				
0	0	0			c2:	**1** **1** 0 0	0.7 (no upd)				
					c4:	0 0 **2** **2**	0.3 (update)			0.7 (no upd)	0.6 (no upd)
1	0	1		0 2 2 0			0.7 (no upd)				
				1 0 0 1			0.2 (update)	0.8 (no upd)			0.6 (no upd)
1	0	1	c1:	0 **2** **2** 0			0.6 (no upd)				
			c3:	**1** 0 0 **1**			0.8 (no upd)				

c1 or c3 = (x2 &^x1) or (x1 & ^x2)   **c2 or c4 = (x1 & x1) or (^x1 & ^x2)**

disjunction of conjunctions of top 2 literals from each clause (for n literals, top n/2 literals)

Figure 10: Using probabilistic update of individual literal inclusion counters

Now we are ready to introduce *the Tsetlin Machine*, which we will do in the next section.

1.4 The Tsetling Machine

The background that we covered in the previous section will facilitate the introduction to the Tsetlin Machine. I just outline it below.

<u>Introducing TAs:</u> inclusion counters as TA states, counter increment/decrements as actions, feedbacks as TA rewards and penalties.

<u>Prediction using positive and negative votes</u>

<u>Refining feedback: Type Ia, Type Ib, Type II</u> *(Figure 11)*

<u>Refining probabilistic updates:</u>

pcu_I = (T - max(-T, min(T,votes)))/(2*T);
pcu_II = (T + max(-T, min(T,votes)))/(2*T);
plu_Ia = (s-1)/s;
plu_Ib = 1/s;

<u>Types of TMs</u>

<u>Tuning TM feedback</u> *(Figure 12 on the next page, just an idea. It looks like the tuned feedback better sustains high true noise probabilities. In general, we can tune the feedback grid for specific noise patterns, if we know what kind on noise dominates in the incoming data stream. E.g., 10% of 0-noise, 90% of 1-noise, etc. We can postpone this discussion for now.*)

polarity	***output***	***clause***	***literal***	***FEEDBACK***		
+	0	0	0			
+	0	0	1			
+	0	1	0	+1	Type II	for exluded literals
+	0	1	1			
+	1	0	0	-1	Type Ib	
+	1	0	1	-1	Type Ib	
+	1	1	0	-1	Type Ib	for exluded literals
+	1	1	1	+1	Type Ia	
-	0	0	0	-1	Type Ib	
-	0	0	1	-1	Type Ib	
-	0	1	0	-1	Type Ib	for exluded literals
-	0	1	1	+1	Type Ia	
-	1	0	0			
-	1	0	1			
-	1	1	0	+1	Type II	for exluded literals
-	1	1	1			

Objectives: Encourage memorizing Literal if Literal is 1 and Clause is 1 and Output matches Polarity
Discourage memorizing Literal if either Literal or Clause is 0 and Output matches Polarity
Encourage memorizing Literal if Literal is 0 and Clause is 1 and Output does not match Polarity

<u>For Positive</u>

Type Ia: if y=1 & ce=1 & lit =1, then counter increment (+1)
Type Ib: if y=1 & ce=0 or lit =0, then counter decrement (-1)
Type II: if y=0 & ce=1 & lit=0, then counter increment (+1)

<u>For Negative</u>

Type Ia: if y=0 & ce=1 & lit =1, then counter increment (+1)
Type Ib: if y=0 & ce=0 or lit =0, then counter decrement (-1)
Type II: if y=1 & ce=1 & lit=0, then counter increment (+1)

Figure 11: TM Feedback Grid

Tuning TM Feedback Grid

polarity	***output***	***clause***	***literal***	***FEEDBACK***	***Tuned FB***
+	0	0	0		
+	0	0	1		
+	0	1	0	+1	+1
+	0	1	1		
+	1	0	0	-1	-1
+	1	0	1	-1	-1
+	1	1	0	-1	-1
+	1	1	1	+1	+1
-	0	0	0	-1	+0.5
-	0	0	1	-1	-1
-	0	1	0	-1	-1
-	0	1	1	+1	-0.5
-	1	0	0		
-	1	0	1		+0.5
-	1	1	0	+1	+1
-	1	1	1		

Objectives: Encourage memorizing Literal if Literal matches Clause and Clause matches Output and Output matches Polarity
Discourage memorizing Literal if either Literal does not match Clause and Output, or Clause does not match Output and Output matches Polarity
Encourage memorizing Literal if Literal does not match Clause but matches Output and Output does not match Polarity
Encourage memorizing Literal if Literal is 0 and Clause is 1 and Output does not match Polarity

For Positive Type Ia: if y=1 & ce=1 & lit =1, then counter increment (+1)
Type Ib: if y=1 & ce=0 or lit =0, then counter decrement (-1)
Type II: if y=0 & ce=1 & lit=0, then counter increment (+1)

For Negative Type Ia: if y=0 & ce=0 & lit =0, then counter increment (+0.5)
Type Ib: if y=0 & ce=1 or lit =1, then counter decrement (-1);
if y=0 & ce=1 and lit =1, then counter decrement (-0.5)
Type II: if y=1 & ce=0 & lit=1, then counter increment (+1)
Type II: if y=1 & ce=1 & lit=0, then counter increment (+1)

Performance impact from Tuning (***FEEDBACK*** vs ***TunedFB***)

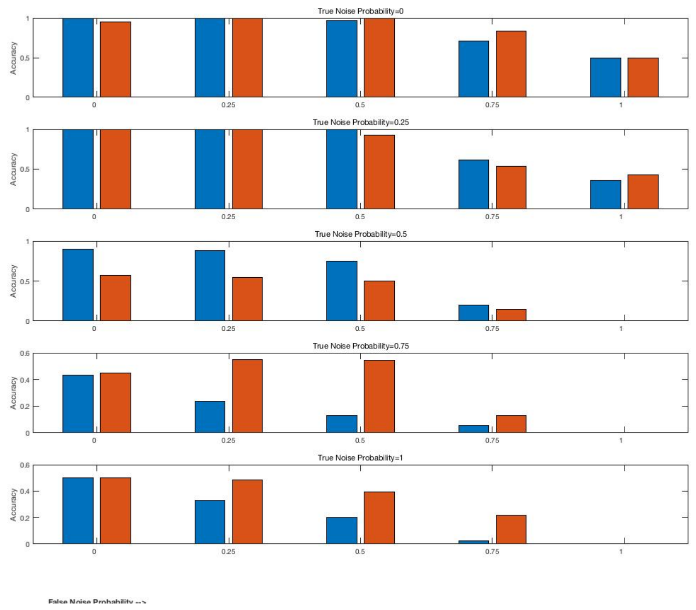


Figure 12: Tuning TM Feedback