Delirious Representations: Enhancing Predictive Systems with Flexible Numeric and Symbolic Domain Integration

Bernhard Gstrein
University of Freiburg
Freiburg, Germany
gstrein@cs.uni-freiburg.de

Armin Biere
University of Freiburg
Freiburg, Germany
biere@cs.uni-freiburg.de

Abstract*—Predictive systems for low-spec devices are crucial in the expanding IoT market. Gradient-based optimization algorithms excel in predictive performance, while logical reduction algorithms offer computational efficiency. Traditional conversion of numeric classifiers to symbolic structures is irreversible, preventing further optimization through gradient-based methods. We introduce a flexible "delirious" representation framework, allowing seamless switching between symbolic and numeric domains. Our approach employs lookup tables with additional flexibility compared to previous methods. We write lookup tables as logic formulas and relax the logical values from binary to the range [0, 1], enabling gradients to traverse the lookup table - a well-defined symbolic structure. We demonstrate the usefulness of a delirious representation on the WiSARD classification system, which incorporates lookup tables at its core, and conduct additional experiments. This originally symbolic-only architecture can now be trained using numeric methods like backpropagation and optimized with logical algorithms post-training. Our results reveal that learning via backpropagation outperforms the original algorithm, which does not utilize backpropagation. We conclude by discussing other possible use cases of delirious representations.

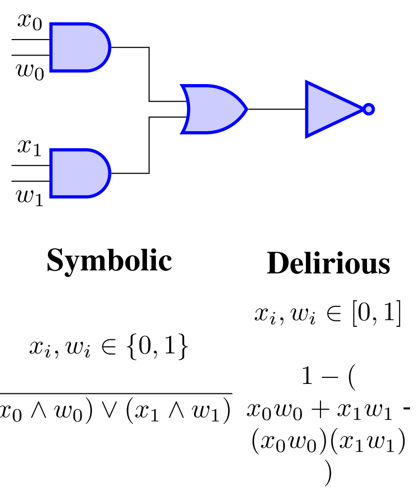

Fig. 1. We introduce a "delirious" framework, where formulas exist in numeric and boolean spaces at the same time.

I. Introduction

Compute-efficient implementations of predictive systems are of great interest today. There is an emerging paradigm, *edge computing*, where deployment on low-spec devices is the task [1]. The usual workflow is to conceptualize particular architectures that are very small and train them using gradient descent. They are often converted to symbolic structures to make these even smaller and more compute-efficient, and logic reduction algorithms optimize them even more. Once converted, however, the models live in a symbolic domain and thus are out of reach for gradient-based optimization.

We introduce a "delirious" classifier framework, simultaneously symbolic and sub-symbolic, benefiting from gradient optimization and logic reduction. We use lookup tables (LUTs) as candidates for such representation. By writing a lookup table as a logic formula and then relaxing the binary values to the floating point range [0, 1], we can pass gradients through a lookup table while still being a well-defined symbolic structure. Figure 1 visualizes a delirious representation of a formula.

Recently there has been surging interest in combining machine learning with symbolic methods to develop optimized classifiers [2, 3, 4]. The lookup operation ignited an interest in replacing the floating-point computation as a computationally cheap alternative [5, 6]. A neural network uses floating-point operations at its core, which are costly. A lookup table does not need any floating-point computation. A significant challenge remained the learning; having no access to gradient-based optimization proved searching for a good model difficult. We seek to alleviate this by the use of a delirious framework.

The architecture that we start with is WiSARD (Wilkie, Stonham & Aleksander's Recognition Device) [7]. It is a sparse weightless neural network with just one hidden layer. It operates on binary values, so there is no floating-point computation, and each neuron is a lookup table. We choose this architecture because the baseline performance is quite good, which is a good starting point. In this work, we show that training by backpropagation [8, 9] vastly outperforms training by the traditional algorithm.

Our contributions are thus as follows:

- We identify delirious operations, i.e., operations suitable for numeric and symbolic computation simultaneously,
- demonstrate the effectiveness of a delirious representation on the WiSARD classifier system and
- discuss possible use cases of a delirious representation.

Section II overviews similar approaches, especially concerning lookup tables. Our core idea is in Section III, where we introduce our delirious framework. We explain in detail how WiSARD works and how it can be made differentiable in Section IV. In Section V, we benchmark our scheme on several datasets. Lastly, we identify in Section VI the main challenges for the future.

II. Related Work

Our work takes inspiration from several places. Using many elements that are already there, we can conceptualize our framework.

The growing IoT market ignited lots of interest in miniature models. There have been a lot of advances in the field of binary neural networks. A good overview is in [10]. For optimization and deployment, the last step often involves converting these binary networks into symbolic representations [11, 12]. Once at a symbolic representation, the model becomes oblivious to backpropagation.

The International Workshop on Logic & Synthesis held a programming contest on synthesizing circuit-based classifiers [2]. The idea was to bridge the gap between numeric and symbolic methods. The final output should be an and-inverter graph (AIG) representing binary classifiers. The following year, the task was image classification, CIFAR-10. The winning team obtained 57% with one million AIG nodes [13]. Although the mission was to bridge the gap between numeric and symbolic methods, there was always a "hard" cutoff between the two.

Concerning lookup tables, they have recently been gaining attention as a low-cost replacement for a neuron in an artificial neural network. An interesting idea to make a network out of lookup tables is here [5]. The original research question was not to excel on a dataset but to answer why neural networks generalize. Still, the developed architecture is a lookup-table-based classifier that somewhat generalizes and could be seen as proof that lookup tables can replace conventional neurons.

Writing logic formulas as an arithmetic expression has attracted interest recently [14, 15]. We can see a scheme integrating numeric and symbolic representations of a particular type of lookup table for the first time in the paper *Deep Differentiable Logic Gate Networks* [16]. That paper demonstrates training two-sparse neural networks whose neurons are lookup tables. With two-sparse, we mean that the input size to each lookup table must be two. The method does not allow for arbitrary input size, but only two; 2-LUTs are made differentiable by relaxing binary variables to the range [0, 1] and writing them as arithmetic formulas. This scheme performs well in terms of accuracy.

The paper *Deep Differentiable Logic Gate Networks* does not mention WiSARD, but the method is very similar. WiSARD only uses one hidden layer, and the arity can be arbitrary, not just two. Using several heuristics, [17] have improved WiSARD concerning accuracy and model size. There are multiple derivative architectures of WiSARD which we will not mention here for the sake of brevity. Unfortunately, to our knowledge, no benchmarking exists comparing conventional WiSARD to derivatives to make a definitive statement on their power. A decent overview of different WiSARD derivatives is in [18, 19].

We identify operations that are delirious, i.e., operations that are suitable for numeric and symbolic computation at the same time. Good examples are the lookup operation, an argmax, or boolean logic. Having identified what is delirious and how the transformation works, we demonstrate the effectiveness of using a delirious representation on WiSARD.

III. Delirious Operations

We can use two main tricks to make boolean logic differentiable, as outlined in [14]. First, let us write the logical functions AND, OR, and NOT as functions in the integer domain. The boolean values True and False will thus become 0 and 1. So $\overline{x_0}$ becomes $1 - x_0$, $x_0 \wedge x_1$ becomes $x_0 \cdot x_1$ and $x_0 \vee x_1$ becomes $x_0 + x_1 - x_0 \cdot x_1$.

The variables are still either 0 or 1 and thus not differentiable. If we relax the range of the variables to [0, 1], we obtain differentiability. Also note that upon input of variables in the range [0, 1], the output will always be in [0, 1].

Having established that logic can be made differentiable, we turn our attention to learning. Even if a circuit is differentiable, without parameters to adjust, no learning happens. That is why lookup tables are a great candidate. They have internal parameters, and learning means adjusting these parameters. Table I shows a 2-LUT.

TABLE I
Lookup-table with arity two.

x_0	x_1	f
0	0	a
0	1	b
1	0	c
1	1	d

Let us now work towards making a lookup table differentiable. To give proper credit, we will show the method from [16] to make 2-LUTs differentiable. We will then depart from solely LUTs with arity two and show how to make LUTs with arbitrary arity differentiable.

A. Weighed Truth Tables Method

One approach to make a lookup table differentiable comes from [16], which considers all truth tables a 2-LUT can represent and calculate simultaneously. If we multiply each truth table output with a weight, the weights become the learnable parameters. After training, we choose the truth table with maximum weight and throw away all other LUTs. We call this method *weighed truth tables*.

Consider Table I, which represents a 2-LUT. The inputs x_0 and x_1 are in binary, as well as the variables a, b, c, and d. We can consider for all LUTs in the classifier all possible truth tables that they can represent and weigh the output of each truth table. So for each input (x_0, x_1), we calculate $x_0 \wedge x_1$, $x_0 \vee x_1$, True, x_1 corresponding to truth tables 0001, 0111, 1111, 0101 and so on. We must do this for all 16 truth tables. According to each truth table, upon being indexed by x_0 and x_1, one value is produced, either 0 or 1. We obtain 16 values and multiply each value i with weight i. In the end, we sum up:

$$a(x_0, x_1) = \sum_{i=1}^{16} \text{LUT}_i(x_0, x_1) w_i, \tag{1}$$

where we denote a as the activation.

For completeness, see Table II for all possible truth tables for a 2-LUT.

TABLE II
ALL POSSIBLE TRUTH TABLES FOR A 2-LUT.

Truth Table	Logical	Arithmetic
0000	False	0
0001	$x_0 \wedge x_1$	$x_0 x_1$
0010	$x_0 \wedge \overline{x_1}$	$x_0 - x_0 x_1$
0011	x_0	x_0
0100	$\overline{x_0} \wedge x_1$	$x_1 - x_0 x_1$
0101	x_1	x_1
0110	$(x_0 \wedge \overline{x_1}) \vee (\overline{x_0} \wedge x_1)$	$x_0 + x_1 - x_0 x_1(3 - x_0 - x_1 + x_0 x_1)$
0111	$x_0 \vee x_1$	$x_0 + x_1 - x_0 x_1$
1000	$\overline{x_0} \wedge \overline{x_1}$	$1 - x_0 - x_1 + x_0 x_1$
1001	$(x_0 \wedge x_1) \vee (\overline{x_0} \wedge \overline{x_1})$	$1 - x_0 - x_1 + x_0 x_1(1 + x_0 + x_1 - x_0 x_1)$
1010	$\overline{x_1}$	$1 - x_1$
1011	$x_0 \vee \overline{x_1}$	$1 - x_1 + x_0 x_1$
1100	$\overline{x_0}$	$1 - x_0$
1101	$\overline{x_0} \vee x_1$	$1 - x_0 + x_0 x_1$
1110	$\overline{x_0} \vee \overline{x_1}$	$1 - x_0 x_1$
1111	True	1

One problem with this approach is that the arity must be strictly two. The number of possible truth tables increases double exponentially. An arity of three with $2^{2^3} = 256$ truth tables would be feasible, albeit slow. An arity of four would already require computing $2^{2^4} = 65536$ truth tables. We know from [20] that the theoretical predictive strength of WiSARD grows exponentially with increasing arity. It makes sense to try out higher arities, so we need a new scheme to represent lookup tables.

B. Arithmetic Lookup Method

We will now demonstrate how to turn a lookup table of arbitrary arity into an arithmetic formula. We call our method *arithmetic lookup*.

Let us demonstrate our method on lookup tables of arity two. Consider Table I. Depending on the incoming pattern from the dataset, a, b, c, or d is the final output from the lookup table. We can write this lookup table as a logic formula:

$$\begin{aligned} &(\overline{x_0} \wedge \overline{x_1} \wedge a) \\ \vee\ &(\overline{x_0} \wedge x_1 \wedge b) \\ \vee\ &(x_0 \wedge \overline{x_1} \wedge c) \\ \vee\ &(x_0 \wedge x_1 \wedge d). \end{aligned} \tag{2}$$

To turn Formula 2 into an arithmetic one, we use two tricks we already used. The first one is, as before, to write the logic values as 0 and 1 - natural numbers. The logical operators thus become

$$\text{NOT}(x_0) = -x_0, \tag{3}$$
$$\text{AND}(x_0, x_1) = x_0 x_1, \tag{4}$$
$$\text{OR}(x_0, x_1) = x_0 + x_1 - x_0 x_1. \tag{5}$$

The second trick is to relax the values 0 and 1 to the range [0, 1]. That way, we can take gradients. What we now can do is instead of computing all 16 truth tables and weighing them, we rewrite Formula 2 using the operators 3, 4 and 5. The lookup table is now a differentiable arithmetic formula. We resolve the operations from right to left, resulting in

$$\begin{aligned} &\text{OR}(\text{OR}(\text{OR}(\text{AND}(\text{AND}(\text{NOT}(x_0), \text{NOT}(x_1)), a), \\ &\quad \text{AND}(\text{AND}(\text{NOT}(x_0), x_1), b)), \\ &\quad \text{AND}(\text{AND}(x_0, \text{NOT}(x_1)), c)), \text{AND}(\text{AND}(x_0, x_1), d)). \end{aligned} \tag{6}$$

We have thus reduced the number of trainable parameters from 16 to 4 for a 2-LUT which is an exponential reduction. A 4-LUT would now have 16 parameters instead of 65536, making it computationally feasible.

What still needs to be solved is that the formula itself snowballs with increasing arity, i.e., there will be many nested expressions. The Python interpreter gives up on our machine's formulas for arity eight and more. An easy fix is to detect sub-expressions and introduce variables for them, similar to a Tseitin transformation [21]. Introducing a new variable for each NOT makes sense, so we avoid computing the same thing multiple times. For AND and OR, we just set a fixed number of allowed recursions, i.e., how many AND are allowed to be in the outer-most AND. For example, if we consider Formula 6 and set the allowed recursion for AND and OR to two, then we obtain

$$\begin{aligned} v_0 &= \text{NOT}(x_0), \\ v_1 &= \text{NOT}(x_1), \\ v_2 &= \text{AND}(\text{AND}(v_0, v_1), a), \\ v_3 &= \text{AND}(\text{AND}(v_0, x_1), b), \\ v_4 &= \text{AND}(\text{AND}(x_0, v_1), c), \\ v_5 &= \text{OR}(\text{OR}(v_2, v_3), v_4), \\ \text{LUT2}(x_0, x_1) &= \text{OR}(v_5, \text{AND}(\text{AND}(x_0, x_1), d)). \end{aligned} \tag{7}$$

We have shown how to make logic and lookup tables of arbitrary arity differentiable. We have considered lookup tables because they have adjustable internal parameters.

What we need now is a classifier architecture. We will use WiSARD [7] because it is promising since it uses lookup tables at its core. We have already shown how to make lookup tables differentiable, so we need minor tweaks to make the whole architecture differentiable. WiSARD comes with its learning algorithm, and we show that the predictive performance improves significantly using backpropagation.

IV. WiSARD

We will now introduce the WiSARD architecture and the learning algorithm that it originally came with. We chose this architecture to make it differentiable because lookup tables are at the core of the classifier - perfect for us.

WiSARD is a classification system successfully commercially used for image classification in the 1980s. Although a rather obscure architecture, the relatively good predictive performance and low computational demand of WiSARD make it worth looking into even today.

Traditionally, WiSARD came with a learning algorithm based on counting relevant patterns in the training dataset and storing them in the lookup tables. Although this algorithm works reasonably well, it must be more potent for many tasks. Consequently, there are lots of derivative architectures and learning schemes that were successful to a limited degree.

We present a scheme to train WiSARD using the backpropagation algorithm. We have already seen how to make lookup tables differentiable, and we use several tricks for the rest of the WiSARD classifier. We go back to the symbolic representation for inference. We show that using backpropagation as a learning algorithm for WiSARD vastly improves performance. For completeness, we also demonstrate how the original WiSARD learning algorithm works.

A. How WiSARD Works

The input is a d-dimensional vector that has binary entries. The corresponding label is a class, so if there are k classes, the label can take values $y \in \{0, \dots, k-1\}$. There are k discriminators. Each discriminator's responsibility is to produce a signal corresponding to the confidence that its class is present. So ideally, if an example has class 2, discriminator 2 has the most potent response to the input, resulting in class 2 being the final prediction.

Each discriminator consists of n lookup tables. Each lookup table has a specific arity. For example, if a lookup table has an arity of three, it takes three inputs. The output of each lookup table is either 0 or 1. Which inputs each lookup table takes are randomly set before training and then fixed forever.

For prediction, each lookup table produces a value, either 0 or 1. For each of the k discriminators, we sum up the values of its lookup tables. Thus, every discriminator produces an integer value. The discriminator with the highest value "wins", meaning class j is the final prediction if discriminator j won. Figure 2 shows an illustration of WiSARD.

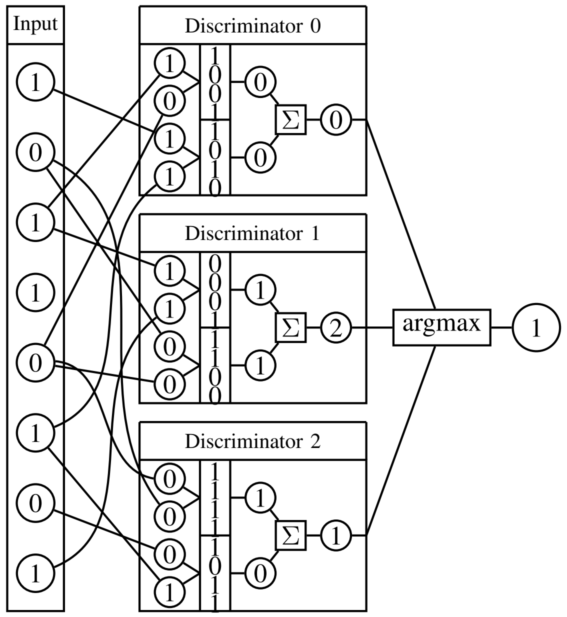

Fig. 2. Illustration of the WiSARD classifier based on computationally efficient lookup tables. The original learning algorithm does not use gradients and thus has its limits. With our framework, the representation of this classifier becomes "delirious", and we can optimize it using sub-symbolic and symbolic methods.

B. Classical Training Algorithm

The classical training algorithm that WiSARD comes with performs reasonably well but has its limits. For the sake of completeness, we briefly explain it.

At first, the lookup-table entries can take arbitrary positive integer values and not only binary. We feed an input example into WiSARD. Each lookup table takes inputs, and the addressed entry is incremented by 1. After providing all training examples to WiSARD, we perform "bleaching", i.e., select a threshold i. We set all entries with $< i$ to 0 and all with $\geq i$ to 1. We evaluate the accuracy of the classifier using different thresholds and choose the one which performs best. A visualization in Figure 3 demonstrates the learning algorithm.

A common problem of this algorithm is *saturation*. It is possible that a lookup table sees every pattern during training, and thus all entries will become 1. If this happens for too many lookup tables, the classifier suffers considerably. To alleviate saturation, we can select a higher bleaching threshold or increase the arity of each lookup table - with the increased cost of making the classifier bigger in memory. We will see later that learning with backpropagation does not lead to the saturation problem.

As we mentioned before, the number of truth tables a lookup table can represent increases double exponentially with the arity. The VC dimension is a measure of how powerful a classifier can be [22]. The authors of [20] show that the VC dimension increases exponentially with the arity, making WiSARD very powerful theoretically.

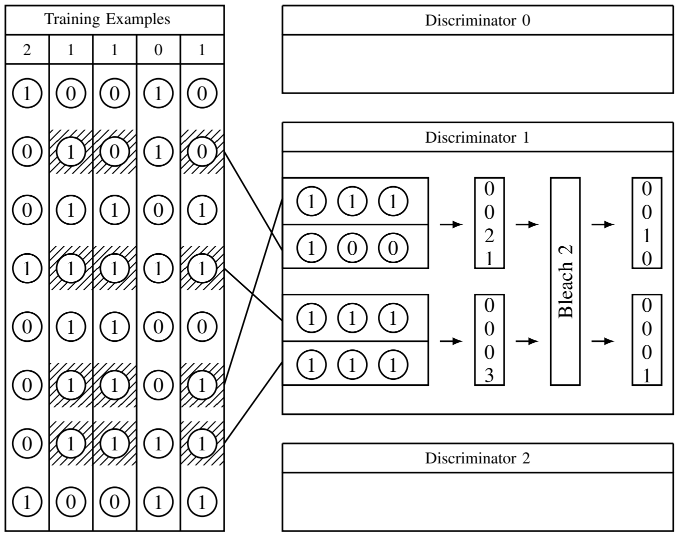

Fig. 3. The WiSARD learning algorithm. Each discriminator only looks at training examples that are relevant to it. For readability, we have left out discriminators 0 and 2.

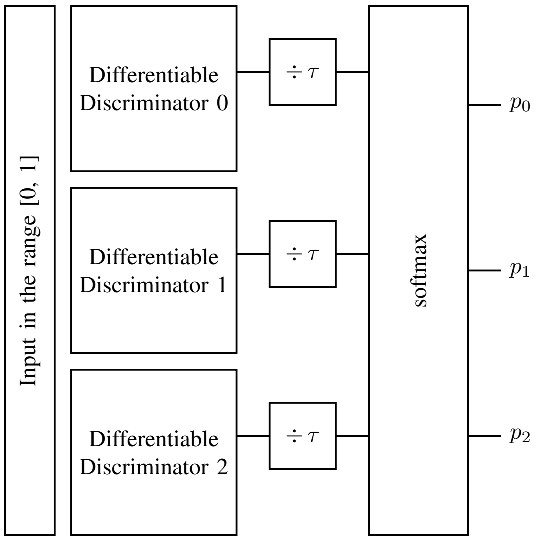
Input in the range [0, 1]
Differentiable Discriminator 0
Differentiable Discriminator 1
Differentiable Discriminator 2
$\div \tau$
softmax
p_0
p_1
p_2

Fig. 4. Modified WiSARD architecture to make it differentiable. Since lookup operations and sum are differentiable, the discriminators are easy. Then we must divide each discriminator output by τ for good convergence. Then, a softmax transforms the values into probabilities. We obtain p_i, the probability for each class i.

C. Making WiSARD Differentiable

Although [16] does not mention WiSARD explicitly, the invented architecture is very similar to WiSARD and differentiable. The difference to WiSARD is that [16] utilizes multiple hidden layers and a fixed arity of two. We will now incorporate key ideas from [16] to render WiSARD differentiable.

We have already shown how to make lookup tables differentiable. For each discriminator, a summing node collects all outputs of the lookup tables. The sum operation is differentiable, and thus the discriminators are differentiable. We will need a few more tricks for the rest of the classifier structure.

After the sum operation, we must divide each output value by a normalizing temperature τ. As [16] have shown, this parameter is crucial for good performance and has to be found heuristically, e.g., by a grid search. Then, we pass each value to a softmax layer. That layer transforms all values into probabilities for each class, which will sum up to 1. That way, the output of differentiable WiSARD is just like that of a neural network, and we can train it. Figure 4 provides a visualization.

After training, we replace the differentiable versions of the discriminator with the non-differentiable ones, get rid of τ and replace the softmax with an argmax.

When training WiSARD using backpropagation, we must use several more tricks to ensure convergence. If we use our scheme of arithmetic lookup, after each update of the lookup table entries, we clip them to the range [0, 1] to prevent them from diverging. We also use "soft labels", i.e., instead of providing labels of 0 and 1 to the loss function, we utilize 0.1 and 0.9. The use of soft labels facilitates convergence and makes learning more stable.

V. Experiments

We run experiments on four different datasets to show the effectiveness of training using backpropagation made possible by using a delirious representation of WiSARD. We perform training using the Python package PyTorch [23]. For arity 2, we evaluate three methods: arithmetic lookup (our), weighed truth tables, and the classical WiSARD algorithm. For arities above 2, we cannot use the weighed truth tables method, so we omit it. The weighed truth tables method becomes impractical for arity three and computationally infeasible for the abovementioned arities. We also use the same number of output neurons as described in [16] to use the same values of τ.

We will also state a baseline performance of a conventional feed-forward neural network for each dataset to get a good overview of our performance. We choose the number of layers and neurons in the neural networks heuristically depending on the input size. We select the number of neurons for the WiSARD classifier heuristically.

We binarize each dataset. The exact binarization method for each dataset differs, and we describe it for each. As for the architecture, i.e., the connections to the lookup tables, we fix each configuration of arities and neurons.

A. Cybersecurity

In computer networks, a packet is a small unit of information containing the actual data and other information, such as source and destination. If someone tries to intrude into the network, we would like to classify good packets from malicious packets. The UNSW-NB15 network data set [24] was created to benchmark classifiers that detect whether a packet is good or evil. Each training example has multiple types of features mixed, such as integers, floats, and strings. We use the binarized version of the UNSW-NB15 dataset [25].

We choose a subset of 60000 training examples. There are 593 binary features and two classes. Table III shows our results. The rows represent the arity, and the columns the number of neurons. For arity two, we report the arithmetic lookup method (ours), the weighed truth tables method from [16], and WiSARD's original learning algorithm. We report just our method and WiSARD's algorithm for arities above two. The baseline neural network with 128 neurons in the first layer and 128 in the second has an accuracy of 86.87%.

TABLE III
TESTING ACCURACIES ON THE CYBERSECURITY DATASET
a : ARITY n : NUMBER OF NEURONS
ARITHMETIC LOOKUP / WEIGHED TRUTH TABLES / WISARD LEARNING ALGORITHM

a \ n	8k	64k	128k
2	89.72 / 86.84 / 75.99	**90.93** / 89.41 / 82.22	90.21 / 90.00 / 82.85
4	89.00 / —— / 82.18	90.00 / —— / 84.57	89.63 / —— / 84.32
8	87.73 / —— / 84.18	87.13 / —— / 84.01	87.20 / —— / 83.88

B. MNIST

The MNIST dataset is an image classification task where digits from 0-9 have to be distinguished [26]. There are 60000 training images and 10000 testing images. Each image is gray-scale and has a dimension of 28x28. We binarize the images by setting each pixel above zero to 1. Each input thus has a size of 784. Table IV shows our results. The baseline neural network with 256 neurons in the first layer and 128 in the second has an accuracy of 98.00%.

Using backpropagation significantly improved training accuracy.

C. CIFAR-10

The CIFAR-10 dataset is color images of ten classes [27]. Each image has a dimension of 32x32 and three channels. We binarize the dataset by expanding the integer value of each pixel into four bins or four bits. Each bit represents in which range the integer is. The bins are thus 0-63, 64-127, 128-191, and 192-255. If, for example, a pixel has a value of 100, then the bit representation would be 0100. The input dimension to the classifier is thus $32 \cdot 32 \cdot 3 \cdot 4 = 12288$. Table V shows our results. The baseline neural network with 1024 neurons in the first layer and 512 in the second has an accuracy of 61.25%.

Again we see a significant improvement in accuracy when using backpropagation. The theoretical learning capacity of WiSARD is vast, as shown by [20]. The learning capacity grows exponentially with increasing arity. Increasing the arity improved our results, but we ask ourselves whether there is room for more improvement.

TABLE IV
TESTING ACCURACIES ON THE MNIST DATASET
a : ARITY n : NUMBER OF NEURONS
ARITHMETIC LOOKUP / WEIGHED TRUTH TABLES / WISARD LEARNING ALGORITHM

a \ n	8k	64k	128k
2	92.31 / 92.13 / 63.73	95.00 / 93.51 / 64.86	95.09 / 93.75 / 65.26
4	95.97 / —— / 77.45	97.18 / —— / 78.82	**97.24** / —— / 78.51
8	97.42 / —— / 88.03	97.83 / —— / 88.81	97.23 / —— / 88.80

TABLE V
TESTING ACCURACIES ON THE CIFAR-10 DATASET
a : ARITY n : NUMBER OF NEURONS
ARITHMETIC LOOKUP / WEIGHED TRUTH TABLES / WISARD LEARNING ALGORITHM

a \ n	8k	64k	128k
2	40.24 / 38.05 / 17.91	46.27 / 45.97 / 17.30	48.58 / 47.50 / 18.08
4	44.65 / —— / 22.92	52.48 / —— / 23.52	**53.91** / —— / 23.83
8	48.76 / —— / 34.46	50.53 / —— / 35.48	49.86 / —— / 36.00

VI. FUTURE WORK

To give a concrete example, we made lookup tables central in this paper and discussed the WiSARD classifier. However, we can use a delirious representation for all sorts of logical expressions.

We have shown in this paper how to make a symbolic classifier into a delirious structure. However, it is also interesting to go the other way around: convert a numeric format into a delirious one. For example, making a conventional neural network delirious would benefit us from efficient training and being able to apply symbolic algorithms simultaneously.

Using delirious representations, methods from multiple domains open up. For example, there is a pruning algorithm for WiSARD that works by removing lookup tables after training [28]. The size of the model shrinks, but with a loss in accuracy. Using a delirious scheme, we can retrain WiSARD after being pruned to recover some accuracy and possibly prune again. We repeat this process until the loss of accuracy gets too high.

To follow up on the IWLS challenge 2021, turning a trained WiSARD classifier into an and-inverter graph [29] would be interesting. The winning team synthesized an AIG with one million nodes with an accuracy of 57% on CIFAR-10 [13].

For image classification, the convolution operation is ideal. An important next step would be to develop a delirious representation of a convolution operation to open the door to image classification in general.

VII. CONCLUSION

We proposed a "delirious" representation scheme, where predictive models are both numeric and symbolic simultaneously. At the heart of our strategy lies making logic differentiable by relaxing binary values to the range [0, 1] and replacing the basic logical operations AND, OR and NOT by numeric ones. We showed the effectiveness of our approach on a traditionally symbolic-only architecture; WiSARD uses lookup tables at its core and has its learning algorithm. We showed how to make lookup tables differentiable, along with the rest of WiSARD, and conducted several experiments. In each, we could see that learning with backpropagation, a numeric algorithm, performs better in learning. We discussed several more use cases of a delirious scheme, especially making numeric structures symbolic and rendering the convolution operation delirious.

References

[1] Keyan Cao et al. “An overview on edge computing research”. In: *IEEE access* 8 (2020), pp. 85714–85728.

[2] Shubham Rai et al. “Logic synthesis meets machine learning: Trading exactness for generalization”. In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2021, pp. 1026–1031.

[3] Jiayuan Mao et al. “The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision”. In: *arXiv preprint arXiv:1904.12584* (2019).

[4] Robin Manhaeve et al. “Deepproblog: Neural probabilistic logic programming”. In: *advances in neural information processing systems* 31 (2018).

[5] Satrajit Chatterjee. “Learning and memorization”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 755–763.

[6] Satrajit Chatterjee and Alan Mishchenko. “Circuit-based intrinsic methods to detect overfitting”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 1459–1468.

[7] Igor Aleksander, WV Thomas, and PA Bowden. “WISARD· a radical step forward in image recognition”. In: *Sensor review* 4.3 (1984), pp. 120–124.

[8] Seppo Linnainmaa. “The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors”. PhD thesis. Master’s Thesis (in Finnish), Univ. Helsinki, 1970.

[9] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[10] Amir Gholami et al. “A survey of quantization methods for efficient neural network inference”. In: *arXiv preprint arXiv:2103.13630* (2021).

[11] Yaman Umuroglu et al. “LogicNets: Co-designed neural networks and circuits for extreme-throughput applications”. In: *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2020, pp. 291–297.

[12] Javier Duarte et al. “Fast inference of deep neural networks in FPGAs for particle physics”. In: *Journal of Instrumentation* 13.07 (2018), P07027.

[13] IWLS. *2021 IWLS Programming Contest Slides*. https://www.iwls.org/contest/2021/IWLS21_Contest_Slides.pdf. Accessed on March 30, 2023. 2021.

[14] Luciano Serafini and Artur d’Avila Garcez. “Logic tensor networks: Deep learning and logical reasoning from data and knowledge”. In: *arXiv preprint arXiv:1606.04422* (2016).

[15] Emile van Krieken, Erman Acar, and Frank van Harmelen. “Analyzing differentiable fuzzy logic operators”. In: *Artificial Intelligence* 302 (2022), p. 103602.

[16] Felix Petersen et al. “Deep Differentiable Logic Gate Networks”. In: *arXiv preprint arXiv:2210.08277* (2022).

[17] Zachary Susskind et al. “Weightless neural networks for efficient edge inference”. In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 2022, pp. 279–290.

[18] Teresa B Ludermir. “Weightless Neural Models: An Overview”. In: *Women in Computational Intelligence: Key Advances and Perspectives on Emerging Topics* (2022), pp. 335–349.

[19] Igor Aleksander et al. “A brief introduction to weightless neural systems.” In: *ESANN*. Citeseer. 2009, pp. 299–305.

[20] Hugo CC Carneiro et al. “The exact vc dimension of the wisard n-tuple classifier”. In: *Neural computation* 31.1 (2019), pp. 176–207.

[21] G. S. Tseitin. “On the Complexity of Derivation in Propositional Calculus”. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28.

[22] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 1999.

[23] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).

[24] Nour Moustafa and Jill Slay. “UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)”. In: *2015 military communications and information systems conference (MilCIS)*. IEEE. 2015, pp. 1–6.

[25] Tadej Murovic and Andrej Trost. “Massively parallel combinational binary neural networks for edge processing”. In: *Elektrotehniski Vestnik* 86.1/2 (2019), pp. 47–53.

[26] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[27] Alex Krizhevsky, Geoffrey Hinton, et al. “Learning multiple layers of features from tiny images”. In: (2009).

[28] Zachary Susskind et al. “Pruning weightless neural networks”. In: *ESANN 2022 proceedings* (2022).

[29] Armin Biere. “The AIGER and-inverter graph (AIG) format version 20071012”. In: (2007).