Lossless hardening with $\partial\mathbb{B}$ nets

Ian Wright [1]

Abstract

$\partial\mathbb{B}$ nets are differentiable neural networks that learn discrete boolean-valued functions by gradient descent. $\partial\mathbb{B}$ nets have two semantically equivalent aspects: a differentiable soft-net, with real weights, and a non-differentiable hard-net, with boolean weights. We train the soft-net by backpropagation and then 'harden' the learned weights to yield boolean weights that bind with the hard-net. The result is a learned discrete function. Unlike existing approaches to neural network binarization the 'hardening' operation involves no loss of accuracy. Preliminary experiments demonstrate that $\partial\mathbb{B}$ nets achieve comparable performance on standard machine learning problems yet are compact (due to 1-bit weights) and interpretable (due to the logical nature of the learnt functions).

1. Introduction

Neural networks must be differentiable. But differentiability means we cannot directly learn discrete functions, such as logical predicates. We can approximate discrete functions by defining continuous relaxations. This paper explores a different approach: we define differentiable functions that 'harden', without approximation, to discrete functions.

Specifically, $\partial\mathbb{B}$ nets have two aspects: a *soft-net*, which is a differentiable function with real weights, and a *hard-net*, which is a discrete function with boolean weights. Both aspects are semantically equivalent. We train the soft-net as normal, using backpropagation, then 'harden' the learned weights to boolean values, which bind with the hard-net to yield a discrete function with identical predictive performance (see Figure 1).

[1]GitHub, Oxford, UK. Correspondence to: Ian Wright <wrighti@acm.org>.

Published at the Differentiable Almost Everything Workshop of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. July 2023.

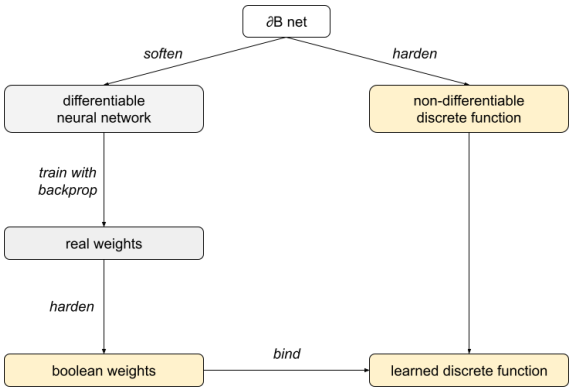

Figure 1. Learning discrete functions with a $\partial\mathbb{B}$ net.

2. $\partial\mathbb{B}$ nets

Definition 2.1 (Soft-bits and hard-bits)**.** A *soft-bit* is a real value in the range $[0, 1]$ and a *hard-bit* is a boolean value from the set $\{0, 1\}$. A soft-bit, x, is *high* if $x > 1/2$, otherwise it is *low*.

A hardening function converts soft-bits to hard-bits.

Definition 2.2 (Hardening)**.** The *hardening* function, $\text{harden}(x_1, \ldots, x_n) = [f(x_1), \ldots, f(x_n)]$, converts soft-bits to hard-bits, where $f(x) = 1$ if $x > 1/2$ and $f(x) = 0$ otherwise.

Soft-nets use soft-bits and hard-nets use hard-bits. A soft-net learns 1-bit weights by representing them, at training time, as real numbers. The equivalent hard-net, at inference time, simply uses 1-bit weights. A soft-net is any differentiable, or differentiable a.e., function, f, that 'hardens' to a hard-net that is a semantically equivalent discrete function, g.

Definition 2.3 (Hard-equivalence)**.** A function, $f : [0, 1]^n \to [0, 1]^m$, is *hard-equivalent* to a discrete function, $g : \{1, 0\}^n \to \{1, 0\}^m$, if $\text{harden}(f(\mathbf{x})) = g(\text{harden}(\mathbf{x}))$ for all $\mathbf{x} \in \{(x_1, \ldots, x_n) \mid x_i \in [0, 1] \setminus \{1/2\}\}$. For shorthand write $f \blacktriangleright g$.

$\partial\mathbb{B}$ nets are composed from 'activation' functions that are hard-equivalent to boolean functions (and natural generalisations).

2.1. Learning to negate

Say we aim to learn to negate a boolean value, x, or leave it unaltered. Represent this decision by a boolean weight, w, where low w means negate and high w means do not. The boolean function that meets this requirement is $\neg(x \oplus w)$. However, this function is not differentiable. Define the differentiable function, $\partial_\neg(w, x) = 1 - w + x(2w - 1)$, where $\partial_\neg(w, x) \blacktriangleright \neg(x \oplus w)$ (see proposition F.1).

Many kinds of differentiable fuzzy logic operators exist (see van Krieken et al. (2022) for a review). So why this functional form? Product logics, where $f(x, y) = xy$ is as a soft version of $x \wedge y$, although hard-equivalent at extreme values, e.g. $f(1, 1) = 1$ and $f(0, 1) = 0$, are not hard-equivalent at intermediate values, e.g. $f(0.6, 0.6) = 0.36$, which hardens to False not True. Gödel-style $\min$ and $\max$ functions, although hard-equivalent over the entire soft-bit range, i.e. $\min(x, y) \blacktriangleright x \wedge y$ and $\max(x, y) \blacktriangleright x \vee y$, are gradient-sparse in the sense that their outputs do not always vary when any input changes, e.g. $\frac{\partial}{\partial x} \max(x, y) = 0$ when $(x, y) = (0.1, 0.9)$. So although the composite function $\max(\min(w, x), \min(1 - w, 1 - x))$ is differentiable and $\blacktriangleright \neg(x \oplus w)$ it does not always backpropagate error to its inputs. In contrast, $\partial_\neg$ always backpropagates error to its inputs because it is a gradient-rich function (see Figure 3).

Definition 2.4 (Gradient-rich)**.** A function, $f : [0, 1]^n \to [0, 1]^m$, is *gradient-rich* if $\frac{\partial f(\mathbf{x})}{\partial x_i} \neq \mathbf{0}$ for all $\mathbf{x} \in \{(x_1, \ldots, x_n) \mid x_i \in [0, 1] \setminus \{1/2\}\}$.

$\partial\mathbb{B}$ nets are composed of hard-equivalent 'activation' functions that are, where possible, gradient-rich. To meet this requirement we introduce the technique of margin packing.

2.2. Margin packing

Say we aim to construct a differentiable analogue of $x \wedge y$. Note that $\min(x, y)$ essentially selects one of x or y as a representative soft-bit that is guaranteed hard-equivalent to $x \wedge y$. However, by selecting only one of x or y then $\min$ is also guaranteed to be gradient-sparse. We define a 'margin packing' method to solve this dilemma.

The main idea of margin packing is (i) select a representative bit that is hard-equivalent to the target discrete function, and then (ii) pack a fraction of the margin between the representative bit and the hard threshold $1/2$ with gradient-rich information. The result is an augmented bit that is a function of all inputs yet hard-equivalent to the target function. More concretely, say a vector of soft-bit inputs $\mathbf{x}$ has an ith element that represents the target discrete function (e.g. if our target is $x \wedge y$ then $\mathbf{x} = [x, y]$ and i is 1 if $x < y$ and $i = 2$ otherwise). Now, if we pack only a fraction of the available margin, $|x_i - 1/2|$, we will not cross the $1/2$ threshold and break the hard-equivalence of the representative bit. The average soft-bit value, $\bar{\mathbf{x}} \in [0, 1]$, is just such a gradient-rich fraction. We therefore define $\text{margin-fraction}(\mathbf{x}, i) = \bar{\mathbf{x}} \times |x_i - 1/2|$. The packed fraction, $\bar{\mathbf{x}}$, of the margin increases or decreases with the average soft-bit value. The available margin, $|x_i - 1/2|$, tends to zero as the representative bit, x_i, tends to the hard threshold $1/2$. At the threshold point there is no margin to pack. Now, define the augmented bit as

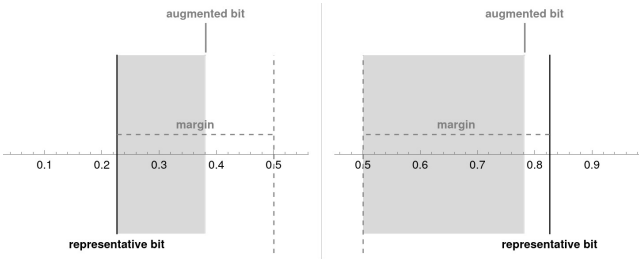

Figure 2. Margin packing for constructing gradient-rich, hard-equivalent functions. A representative bit, z, is hard-equivalent to a discrete target function but gradient-sparse (e.g. $z = \min(x, y) \blacktriangleright x \wedge y$). On the left z is low, $z < 1/2$; on the right z is high, $z > 1/2$. We can pack a fraction of the margin between z and the hard threshold $1/2$ with additional gradient-rich information without affecting hard-equivalence. A natural choice is the mean soft-bit, $\bar{\mathbf{x}} \in [0, 1]$. The grey shaded areas denote the packed margins and the final augmented bit. On the left $\approx 60\%$ of the margin is packed; on the right $\approx 90\%$.

$$\text{augmented-bit}(\mathbf{x}, i) = \begin{cases} 1/2 + \text{margin-fraction}(\mathbf{x}, i) & \text{if } x_i > 1/2 \\ x_i + \text{margin-fraction}(\mathbf{x}, i) & \text{otherwise,} \end{cases} \quad (1)$$

which is differentiable a.e. Note that if the representative bit is high (resp. low) then the augmented bit is also high (resp. low). The difference between the augmented and representative bit depends on the size of the available margin and the mean soft-bit value. Almost everywhere, an increase (resp. decrease) of the mean soft-bit increases (resp. decreases) the value of the augmented bit (see Figure 2). Note that if the ith bit is representative (i.e. hard-equivalent to the target function) then so is the augmented bit (see lemma F.2). We use margin packing, where appropriate, to define gradient-rich, hard-equivalents of boolean functions.

2.3. Differentiable $\wedge$, $\vee$ and $\Rightarrow$

We aim to construct a differentiable analogue of the boolean function $\bigwedge_{i=1}^n x_i$. A representative bit is $\min(x_1, \ldots, x_n)$. The function $\partial_\wedge(\mathbf{x}) = \text{augmented-bit}(\mathbf{x}, \text{argmin}_i\, x[i])$ is therefore hard-equivalent to the boolean function $\bigwedge_{i=1}^n x_i$ (see proposition F.3). In the special case $n = 2$ we get the piecewise function, $\partial_\wedge(x, y) = 1/2 + 1/2(x + y)(\min(x, y) - 1/2)$ if $\min(x, y) > 1/2$, and $\partial_\wedge(x, y) = \min(x, y) + 1/2(x + y)(1/2 - \min(x, y))$ otherwise. Note

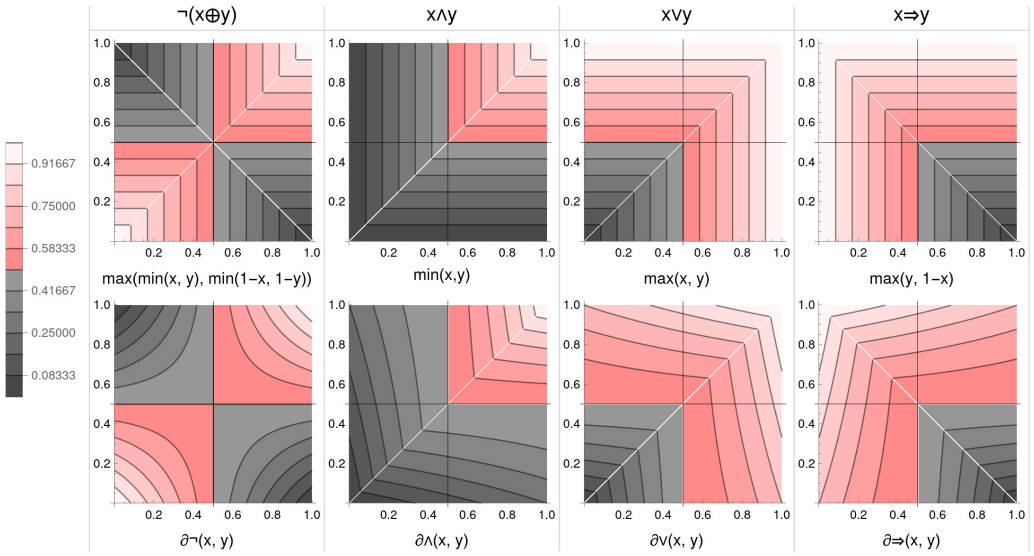

Figure 3. Gradient-rich versus gradient-sparse differentiable boolean functions. Each column contains contour plots of functions $f(x, y)$ that are hard-equivalent to a boolean function (one of $\neg(x \oplus y)$, $x \wedge y$, $x \vee y$, or $x \Rightarrow y$). Every function is continuous and differentiable a.e. (white lines indicate non-continuous derivatives). The upper plots are gradient-sparse, where vertical and horizontal contours indicate the function is constant with respect to one of its inputs, i.e. $\partial f/\partial y = 0$ or $\partial f/\partial x = 0$. The lower plots are gradient-rich, where the curved contours indicate the function always varies with respect to any of its inputs, i.e. $\partial f/\partial y \neq 0$ and $\partial f/\partial x \neq 0$. $\partial\mathbb{B}$ nets use gradient-rich functions to ensure that error is always backpropagated to all inputs.

that $\partial_\wedge$ is differentiable a.e. and gradient-rich (see Figure 3). The differentiable analogue of $\vee$ is identical to $\wedge$, except the representative bit is selected by $\max$. The function $\partial_\vee(\mathbf{x}) = \text{augmented-bit}(\mathbf{x}, \text{argmax}_i\, x[i])$ is hard-equivalent to the boolean function $\bigvee_{i=1}^n x_i$ (see proposition F.4) (see Figure 3). The differentiable analogue of $\Rightarrow$ (material implication) is defined in terms of $\partial_\vee$. The function $\partial_\Rightarrow(x, y) = \partial_\vee(y, 1 - x)$, is hard-equivalent to $x \Rightarrow y$ (see proposition F.5). We can define analogues of all the basic boolean operators in a similar manner.

2.4. Differentiable majority

The boolean majority function is particularly important for tractable learning because it is a threshold function:

$$\text{Maj}(\mathbf{x}) = \left\lfloor \frac{1}{2} + \frac{\sum_{i=1}^n x_i - 1/2}{n} \right\rfloor,$$

where we count False as 0 and True as 1. We aim to construct a differentiable analogue of Maj.

Maj for n bits in DNF form is a disjunction of $\binom{n}{k}$ conjunctive clauses of size k, where $k = \lceil n/2 \rceil$. In principle we can implement a differentiable analogue of Maj in terms of $\partial_\wedge$ and $\partial_\vee$. However, the number of terms grows exponentially with the variables. No general algorithm exists to find the minimal representation of Maj for arbitrary n. Instead, we trade-off time for memory costs. Assume that the function $\text{sort}(\mathbf{x})$ sorts the elements of $\mathbf{x}$ in ascending order. Then the 'median' soft-bit, $\text{majority-index}(\mathbf{x}) = \lceil \frac{|\mathbf{x}|}{2} \rceil$, is representative. Applying margin packing, define the differentiable function $\partial\text{Maj}(\mathbf{x}) = \text{augmented-bit}(\text{sort}(\mathbf{x}), \text{majority-index}(\mathbf{x}))$, which is hard-equivalent to Maj (see theorem F.7). Note that ∂Maj is differentiable a.e. and gradient-rich (see Figure 4). If sort is quicksort then the average time-complexity of ∂Maj is $\mathcal{O}(n \log n)$, which makes ∂Maj more expensive than $\partial_\neg$, $\partial_\wedge$, $\partial_\vee$ and $\partial_\Rightarrow$ at training time. The sort operation could be replaced by the Floyd-Rivest algorithm, which has linear average time complexity (Kiwiel, 2005). However, in the hard $\partial\mathbb{B}$ net we efficiently implement Maj as a discrete program that simply checks if the majority of bits are high. Note that we use sort to define a differentiable function that is exactly equivalent to a discrete function (rather than defining a continuous approximation to sorting, e.g. Cuturi et al. (2019), Grover et al. (2019) and Petersen et al. (2022b)).

We apply this methodology to construct functions that harden to other kinds of boolean functions, such as boolean counting (see Section C). This basic set of functions is sufficient to learn non-trivial relationships from data. $\partial\mathbb{B}$ net layers are compositions of these functions, where composition preserves hard-equivalence (see Sections A and B).

3. Learning discrete functions

We briefly illustrate the kind of discrete program that $\partial\mathbb{B}$ nets can learn. Consider the toy problem of predicting

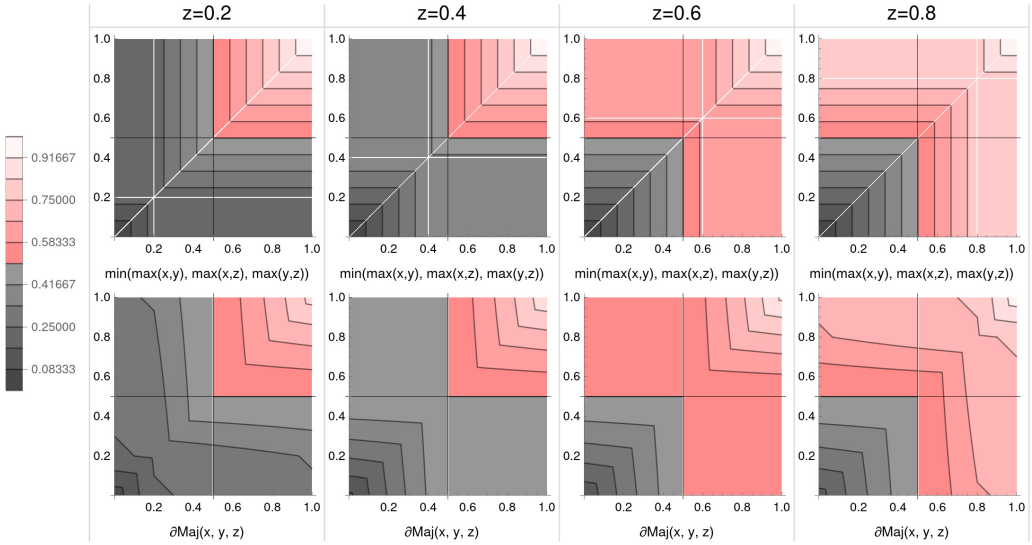

Figure 4. Differentiable boolean majority. The boolean majority function for three variables in DNF form is $\mathrm{Maj}(x, y, z) = (x \wedge y) \vee (x \wedge y) \vee (y \wedge z)$. The upper row contains contour plots of $f(x, y, z) = \min(\max(x, y), \max(x, z), \max(y, z))$ for values of $z \in \{0.2, 0.4, 0.6, 0.8\}$. f is differentiable and $\blacktriangleright\mathrm{Maj}$ but gradient-sparse (vertical and horizontal contours indicate constancy with respect to an input). Also, the number of terms in f grows exponentially with the number of variables. The lower row contains contour plots of $\partial\mathrm{Maj}(x, y, z)$ for the same values of z. $\partial\mathrm{Maj}$ is differentiable and $\blacktriangleright\mathrm{Maj}$ yet gradient-rich (curved contours indicate variability with respect to any inputs). In addition, the number of terms in $\partial\mathrm{Maj}$ is constant with respect to the number of variables.

whether a person wears a t-shirt (label 0) or a coat (label 1) conditional on 5 boolean features (see Table 1).

very-cold	cold	warm	very-warm	outside	label
1	0	0	0	0	1
0	0	0	1	1	1
0	0	1	0	1	0
0	0	0	1	0	0
...	...	...	...	...	...

Table 1. A toy learning problem

We train the $\partial\mathbb{B}$ net described in Figure 55. Once trained we harden the net to a discrete program (see Section D) that generates 2 hard-bits, corresponding to each label. The program symbolically simplifies to:

```
def dbNet(very-cold, cold, warm, very-warm, outside):
return [
4 !very-cold + 4 !cold + (3 warm + !warm) + (very-warm + 3 !very-
      warm) + (outside + 3 !outside) >= 11,
(very-cold + 3 !very-cold) + 4 cold + 4 !warm + (3 very-warm + !
      very-warm) + 2 (outside + !outside) >= 11
]
```

Note that the program linearly weights multiple pieces of evidence due to the presence of the $\partial\mathrm{Maj}$ operator (overkill for this toy problem). We can read-off that the $\partial\mathbb{B}$ net has learned 'if not very-cold and not cold and not outside then wear a t-shirt'; and 'if cold and not (warm or very-warm) and outside then wear a coat' etc. Section E compares $\partial\mathbb{B}$ net performance against other classification algorithms on standard machine learning problems.

4. Related work

Binary neural networks, e.g. Courbariaux et al. (2015), reduce real weights and/or activations to binary, saving model size and inference costs. The binarization step is lossy, which loses accuracy (Qin et al., 2020). Deep differentiable logic gate networks (Petersen et al., 2022a) consist of 2-input neurons arranged in a fixed topology. Each neuron learns a differentiable probability distribution over the 16 possible binary functions. Post-training the neurons are discretized to the most probable binary function. This step is lossy, which loses accuracy. $\partial\mathbb{B}$ nets aim to explore the design space of differentiable nets that enable lossless hardening.

5. Conclusion

$\partial\mathbb{B}$ nets are differentiable nets that are hard-equivalent to non-differentiable, boolean-valued functions. $\partial\mathbb{B}$ nets can therefore learn discrete functions by gradient descent. Ensuring hard-equivalence requires defining new kinds of activation functions and network layers. 'Margin packing' is a potentially general technique for constructing differentiable functions that are hard-equivalent yet gradient-rich. An advantage of $\partial\mathbb{B}$ nets is that 'hardening' to 1-bit weights has provably identical accuracy. At inference time $\partial\mathbb{B}$ nets are highly compact and potentially cheap to evaluate. Preliminary experiments demonstrate that $\partial\mathbb{B}$ nets achieve comparable performance to existing approaches.

Acknowledgements

GitHub Next sponsored this research. Thanks to Pavel Augustinov, Richard Evans, Johan Rosenkilde, Max Schaefer, Ganesh Sittampalam, Tamás Szabó, Albert Ziegler and the anonymous referees for helpful discussions and feedback.

References

Bengio, Y., Léonard, N., and Courville, A. C. Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, abs/1308.3432, 2013. URL http://arxiv.org/abs/1308.3432.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pp. 3123–3131, Cambridge, MA, USA, 2015. MIT Press.

Cuturi, M., Teboul, O., and Vert, J.-P. Differentiable ranking and sorting using optimal transport. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/d8c24ca8f23c562a5600876ca2a550ce-Paper.pdf.

Granmo, O.-C. The binary iris dataset. GitHub repository, a. URL https://github.com/cair/TsetlinMachine.

Granmo, O.-C. The noisy XOR dataset. GitHub repository, b. URL https://github.com/cair/TsetlinMachine.

Granmo, O.-C. The Tsetlin machine – a game theoretic bandit driven approach to optimal pattern recognition with propositional logic, 2018. URL https://arxiv.org/abs/1804.01508.

Grover, A., Wang, E., Zweig, A., and Ermon, S. Stochastic optimization of sorting networks via continuous relaxations. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1eSS3CcKX.

Heek, J., Levskaya, A., Oliver, A., Ritter, M., Rondepierre, B., Steiner, A., and van Zee, M. Flax: A neural network library and ecosystem for JAX, 2023. URL http://github.com/google/flax.

Kiwiel, K. C. On Floyd and Rivest's SELECT algorithm. *Theoretical Computer Science*, 347 (1):214–238, 2005. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2005.06.032. URL https://www.sciencedirect.com/science/article/pii/S0304397505004081.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

Petersen, F., Borgelt, C., Kuehne, H., and Deussen, O. Deep differentiable logic gate networks. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 2006–2018. Curran Associates, Inc., 2022a. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/0d3496dd0cec77a999c98d35003203ca-Paper-Conference.pdf.

Petersen, F., Borgelt, C., Kuehne, H., and Deussen, O. Monotonic differentiable sorting networks. In *International Conference on Learning Representations*, 2022b. URL https://openreview.net/forum?id=IcUWShptD7d.

Qin, H., Gong, R., Liu, X., Bai, X., Song, J., and Sebe, N. Binary neural networks: A survey. *Pattern Recognition*, 105:107281, 2020. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2020.107281. URL https://www.sciencedirect.com/science/article/pii/S0031320320300856.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

van Krieken, E., Acar, E., and van Harmelen, F. Analyzing differentiable fuzzy logic operators. *Artificial Intelligence*, 302:103602, 2022. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2021.103602. URL https://www.sciencedirect.com/science/article/pii/S0004370221001533.

A. Boolean logic layers

The full variety of $\partial\mathbb{B}$ net architectures is to be explored. Here we define basic layers sufficient for the classification experiments.

A $\partial_\neg\mathrm{Layer}$ of width n learns to negate up to n different subsets of the elements of its input vector:

$$\partial_\neg\mathrm{Layer} : [0,1]^{n\times m} \times [0,1]^m \to [0,1]^{n\times m},$$
$$(\mathbf{W}, \mathbf{x}) \mapsto \begin{bmatrix} \partial_\neg(w_{1,1}, x_1) & \dots & \partial_\neg(w_{1,m}, x_m) \\ \vdots & \ddots & \vdots \\ \partial_\neg(w_{n,1}, x_1) & \dots & \partial_\neg(w_{n,m}, x_m) \end{bmatrix}$$

where $\mathbf{x}$ is a soft-bit input vector, $\mathbf{W}$ is a weight matrix and n is the layer width. Similarly, A $\partial_\Rightarrow\mathrm{Layer}$ of width n learns to 'mask to true or nop' up to n different subsets of the elements of its input vector:

$$\partial_\Rightarrow\mathrm{Layer}(\mathbf{W}, \mathbf{x}) = \begin{bmatrix} \partial_\Rightarrow(w_{1,1}, x_1) & \dots & \partial_\Rightarrow(w_{1,m}, x_m) \\ \vdots & \ddots & \vdots \\ \partial_\Rightarrow(w_{n,1}, x_1) & \dots & \partial_\Rightarrow(w_{n,m}, x_m) \end{bmatrix}.$$

A $\partial_\wedge\mathrm{Neuron}$ learns to logically $\wedge$ a subset of its input vector:

$$\partial_\wedge\mathrm{Neuron} : [0,1]^n \times [0,1]^n \to [0,1],$$
$$(\mathbf{w}, \mathbf{x}) \mapsto \min(\partial_\Rightarrow(w_1, x_1), \dots, \partial_\Rightarrow(w_n, x_n)),$$

where $\mathbf{w}$ is a weight vector. Each $\partial_\Rightarrow(w_i, x_i)$ learns to include or exclude x_i from the conjunction depending on weight w_i. For example, if $w_i > 0.5$ then x_i affects the value of the conjunction since $\partial_\Rightarrow(w_i, x_i)$ passes-through a soft-bit that is high if x_i is high, and low otherwise; but if $w_i \leq 0.5$ then x_i does not affect the conjunction since $\partial_\Rightarrow(w_i, x_i)$ always passes-through a high soft-bit. A $\partial_\wedge\mathrm{Layer}$ of width n learns up to n different conjunctions of subsets of its input (of whatever size). A $\partial_\vee\mathrm{Neuron}$ is defined similarly:

$$\partial_\vee\mathrm{Neuron} : [0,1]^n \times [0,1]^n \to [0,1],$$
$$(\mathbf{w}, \mathbf{x}) \mapsto \max(\partial_\wedge(w_1, x_1), \dots, \partial_\wedge(w_n, x_n)).$$

Each $\partial_\wedge(w_i, x_i)$ learns to include or exclude x_i from the disjunction depending on weight w_i. A $\partial_\vee\mathrm{Layer}$ of width n learns up to n different disjunctions of subsets of its input (of whatever size).

We can compose $\partial_\neg$, $\partial_\wedge$ and $\partial_\vee$ layers to learn boolean formulae of arbitrary width and depth.

B. Classification layers

In classification problems the final layer of a neural network is typically interpreted as a vector of real-valued logits, one for each label, where the index of the maximum logit indicates the most probable label. However, we cannot interpret a soft-bit vector as logits without violating hard-equivalence. In addition, when training $\partial\mathbb{B}$ nets, loss functions should be a function of hardened bits, otherwise gradient descent may non-optimally traverse trajectories that take no account of the hard threshold at $1/2$. For example, consider that an instance is correctly classified by a 1-hot vector with high bit $x = 0.51$. Updating the net's weights to change this value to $0.51 + \epsilon$ will not improve accuracy and may prevent the correct classification of a different instance.

For these reasons, $\partial\mathbb{B}$ nets have a final 'hardening' layer to ensure that loss is a function of hard, not soft, bits:

$$\partial\mathrm{harden} : [0,1]^n \to [0,1]^n,$$
$$\mathbf{x} \mapsto \mathrm{harden}(\mathbf{x}).$$

The harden function is not differentiable and therefore $\partial\mathrm{harden}$ uses the straight-through estimator (Bengio et al., 2013) during backpropagation. By restricting the use of the straight-through estimator to final layers we avoid compounding gradient estimation errors to deeper parts of the network. Note that $\partial\mathrm{harden}$ is hard-equivalent to a nop.

$\partial\mathbb{B}$ nets can re-use many of the techniques deployed in standard neural networks. For example, for improved generalisation, we define a 'boolean' analogue of the dropout layer (Srivastava et al., 2014):

$$\begin{aligned}\partial\text{dropout} : [0,1]^n \times [0,1] &\to [0,1]^n,\\ (\mathbf{x}, p) &\mapsto [f(x_1, p), \ldots, f(x_n, p)],\end{aligned}$$

where

$$f(x,p) = \begin{cases} 1-x, & \text{with probability } p \\ x, & \text{otherwise.} \end{cases}$$

At train time $\partial\text{dropout}$ randomly negates soft-bit values with probability p. At test time, and in the hard-net, $\partial\text{dropout}$ is a nop.

C. Differentiable counting

A boolean counting function $f(\mathbf{x})$ is True if a counting predicate, $c(\mathbf{x})$, holds over its n inputs. We aim to construct a differentiable analogue of $\text{count}(\mathbf{x}, k)$ where $c(\mathbf{x}) := |\{x_i : x_i = 1\}| = k$ (i.e. 'exactly k high'), which can be useful in multiclass classification problems. Define

$$\begin{aligned}\partial\text{count-hot} : [0,1]^n &\to [0,1]^{n+1},\\ \mathbf{x} &\mapsto \text{low-high}(\text{sort}(\mathbf{x})),\end{aligned}$$

where

$$\begin{aligned}&\text{low-high} : [0,1]^n \to [0,1]^{n+1},\\ &\mathbf{x} \mapsto \left[\partial_\wedge(1, x_1), \partial_\wedge(1-x_1, x_2), \ldots, \partial_\wedge(1-x_{n-1}, x_n), \partial_\wedge(1-x_n, 1)\right].\end{aligned}$$

$\partial\text{count-hot}(\mathbf{x})$ outputs a 1-hot vector where the index of the high bit is the number of low bits in $\mathbf{x}$. Note that $\partial\text{count-hot}$ is differentiable, gradient-rich and hard-equivalent to the boolean function

$$\begin{aligned}&\text{count-hot} : \{0,1\}^n \to \{0,1\}^{n+1},\\ &\mathbf{x} \mapsto \left[\text{k-of-n}(\mathbf{x}, 0), \text{k-of-n}(\mathbf{x}, 1), \ldots, \text{k-of-n}(\mathbf{x}, n)\right],\end{aligned}$$

where

$$\text{k-of-n}(\mathbf{x}, k) = \bigvee_{|S|=k} \bigwedge_{i \in S} x_i \bigwedge_{j \notin S} \neg x_j$$

(see proposition F.8). However, in the hard $\partial\mathbb{B}$ net we efficiently implement count-hot as a discrete program that simply counts the number of low bits.

We can construct various kinds of boolean counting functions from $\partial\text{count-hot}$. For example, $\partial\text{count}(\mathbf{x}, k)$ is straightforwardly $\partial\text{count-hot}(\mathbf{x})[k]$ where margin-packing ensures that this single soft-bit is gradient-rich.

D. An example of hard-equivalence

The $\partial\mathbb{B}$ net specified in Figure 5 (with 40 soft-bit weights) has two single-bit output ports and is hard-equivalent to the following discrete program (with 40 hard-bit weights):

```
def dbNet(very-cold, cold, warm, very-warm, outside):
return [
ge(sum((sum((sum((sum((sum((sum((sum((sum((sum((sum((sum((sum((sum((sum((sum((0, not(xor(ne(very-cold, 0), w1)
      ))), not(xor(ne(cold, 0), w2)))), not(xor(ne(warm, 0), w3)))), not(xor(ne(very-warm, 0), w4)))), not(xor(ne(outside, 0), w5)))),
      not(xor(ne(very-cold, 0), w6)))), not(xor(ne(cold, 0), w7)))), not(xor(ne(warm, 0), w8)))), not(xor(ne(very-warm, 0), w9)))), not
      (xor(ne(outside, 0), w10)))), not(xor(ne(very-cold, 0), w11)))), not(xor(ne(cold, 0), w12)))), not(xor(ne(warm, 0), w13)))), not(
      xor(ne(very-warm, 0), w14)))), not(xor(ne(outside, 0), w15)))), not(xor(ne(very-cold, 0), w16)))), not(xor(ne(cold, 0), w17)))),
      not(xor(ne(warm, 0), w18)))), not(xor(ne(very-warm, 0), w19)))), not(xor(ne(outside, 0), w20)))), 11),
ge(sum((sum((sum((sum((sum((sum((sum((sum((sum((sum((sum((sum((sum((sum((sum((0, not(xor(ne(very-cold, 0), w21
      )))), not(xor(ne(cold, 0), w22)))), not(xor(ne(warm, 0), w23)))), not(xor(ne(very-warm, 0), w24)))), not(xor(ne(outside, 0), w25)
      ))), not(xor(ne(very-cold, 0), w26)))), not(xor(ne(cold, 0), w27)))), not(xor(ne(warm, 0), w28)))), not(xor(ne(very-warm, 0), w29
      )))), not(xor(ne(outside, 0), w30)))), not(xor(ne(very-cold, 0), w31)))), not(xor(ne(cold, 0), w32)))), not(xor(ne(warm, 0), w33)
      ))), not(xor(ne(very-warm, 0), w34)))), not(xor(ne(outside, 0), w35)))), not(xor(ne(very-cold, 0), w36)))), not(xor(ne(cold, 0),
      w37)))), not(xor(ne(warm, 0), w38)))), not(xor(ne(very-warm, 0), w39)))), not(xor(ne(outside, 0), w40)))), 11)
]
```

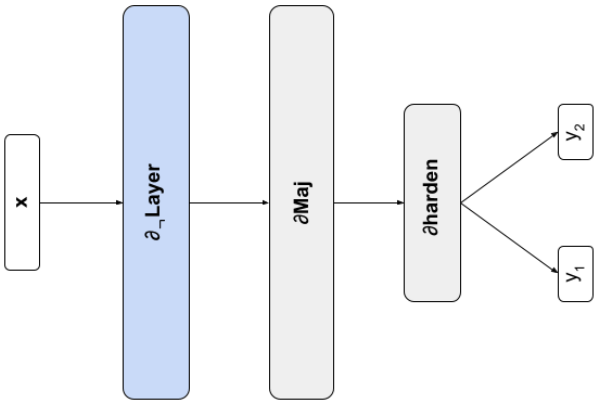

Figure 5. A ∂𝔹 net to illustrate hardening. The net concatenates a $\partial\neg\mathrm{Layer}$ (of width 8) with a reshaping layer that outputs two vectors, which get reduced, by a $\partial\mathrm{Maj}$ operator, to 2 soft-bits, one for each class label. A final $\partial\mathrm{harden}$ layer ensures the loss is a function of hard bits. The net's weights, once hardened, consume 40 bits (5 bytes).

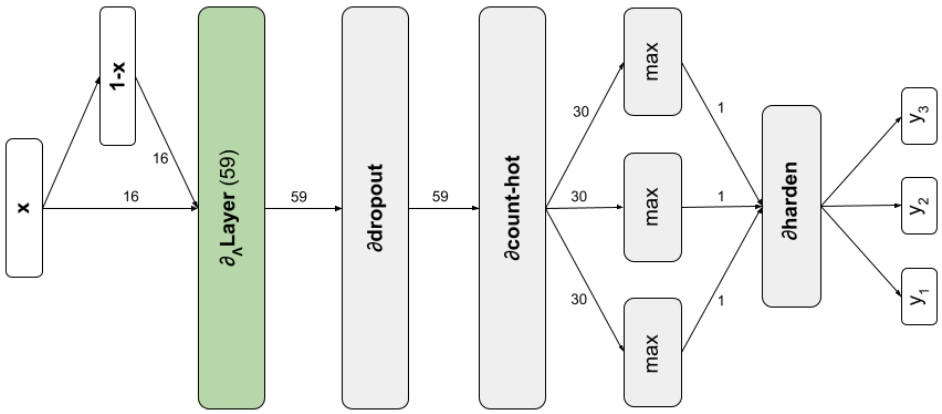

Figure 6. A ∂𝔹 net for the binary Iris problem. The net concatenates the soft-bit input, $\mathbf{x}$ (length 16), with its negation, $\mathbf{1}-\mathbf{x}$, and supplies the resulting vector (length 32) to a $\partial\wedge\mathrm{Layer}$ (width 59), a $\partial\mathrm{dropout}$ layer for improved generalisation, a $\partial\text{count-hot}$ layer that generates a 1-hot vector (width 60) that is reduced by $\max$ to a 1-hot vector of 3 classification bits. A final $\partial\mathrm{harden}$ ensures the loss is a function of hard bits. The net's weights, once hardened, consume 236 bytes.

E. Experiments

The ∂𝔹 net library is implemented in Flax (Heek et al., 2023) and JAX (Bradbury et al., 2018) and available at `github.com/Z80coder/db-nets`. The library supports the specification of a ∂𝔹 net as Python code, which automatically defines (i) the soft-net for training (weights are floats), (ii) a hard-net for inference (weights are booleans), and (iii) a symbolic net for interpretation (weights and inputs are symbols). The symbolic net, when evaluated, interprets its own JAX expression and outputs a description of the discrete program it computes.

We compare the performance of ∂𝔹 nets against standard ML approaches on three problems: the classic Iris dataset, an adversarial noisy XOR problem, and MNIST.

E.1. Binary Iris

The Iris dataset has 150 examples with 4 inputs (sepal length and width, and petal length and width), and 3 labels (*setosa*, *versicolour*, and *virginica*). We use the binary version of the Iris dataset (Granmo, a) where each input float is represented by 4 bits. We perform 1000 experiments, each with a different random seed. Each experiment randomly partitions the data into 80% training and 20% test sets. We initialize the network, described in Figure 6, with all weights $w_i = 0.3$ and train for

	accuracy				
	mean	5 %ile	95 %ile	min	max
Tsetlin	95.0 +/- 0.2	86.7	100.0	80.0	100.0
$\partial\mathbb{B}$	**93.9 +/- 0.1**	**86.7**	**100.0**	**80.0**	**100.0**
neural network	93.8 +/- 0.2	86.7	100.0	80.0	100.0
SVM	93.6 +/- 0.3	86.7	100.0	76.7	100.0
naive Bayes	91.6 +/- 0.3	83.3	96.7	70.0	100.0

Table 2. Binary Iris results measured over 1000 experiments.

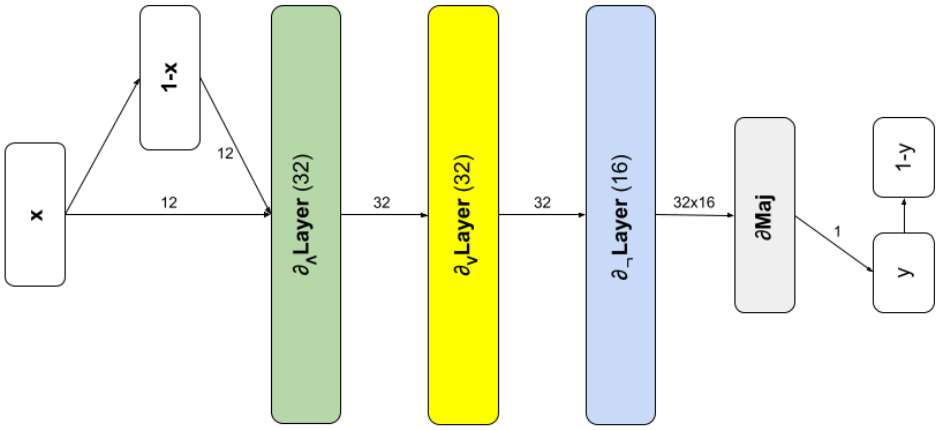

Figure 7. A $\partial\mathbb{B}$ net for the noisy xor problem. The net concatenates the soft-bit input, $\mathbf{x}$ (length 12), with its negation, $\mathbf{1} - \mathbf{x}$, and supplies the resulting vector (length 24) to a $\partial_\wedge$Layer (width 32), $\partial_\vee$Layer (width 32), $\partial_\neg$Layer (width 16), and a final ∂Maj to produce a single soft-bit $y \in [0, 1]$ (to predict odd parity) and its negation $1 - y$ (to predict even parity). The net's weights, once hardened, consume 288 bytes.

1000 epochs with the RAdam optimizer and softmax cross-entropy loss.

We measure the accuracy of the final net to avoid hand-picking the best configuration. Table 2 compares the $\delta\mathbb{B}$ net against other classifiers (Granmo, 2018). Naive Bayes performs the worst. The Tsetlin machine performs best on this problem, with the $\partial\mathbb{B}$ net second.

E.2. Noisy XOR

The noisy XOR dataset (Granmo, b) is an adversarial parity problem with noisy non-informative features. The dataset consists of 10K examples with 12 boolean inputs and a target label (where 0 = odd and 1 = even) that is a XOR function of 2 of the inputs. The remaining 10 inputs are entirely random. We train on 50% of the data where, additionally, 40% of the labels are inverted. We initialize the network described in Figure 7 with random weights distributed close to the hard threshold at $1/2$ (i.e. in the $\partial_\wedge$Layer, $w_i = 0.501 \times b + 0.3 \times (1 - b)$ where $b \sim \text{Bernoulli}(0.01)$; in the $\partial_\vee$Layer, $w_i = 0.7 \times b + 0.499 \times (1 - b)$ where $b \sim \text{Bernoulli}(0.99)$); and in the $\partial_\neg$Layer, $w_i \sim \text{Uniform}(0.499, 0.501)$. We train for 2000 epochs with the RAdam optimizer and softmax cross-entropy loss.

We measure the accuracy of the final net on the test data to avoid hand-picking the best configuration. Table 3 compares the $\partial\mathbb{B}$ net against other classifiers (Granmo, 2018). The high noise causes logistic regression and naive Bayes to randomly guess. The SVM hardly performs better. In contrast, the multilayer neural network, Tsetlin machine, and $\partial\mathbb{B}$ net all successfully learn the underlying XOR signal. The Tsetlin machine performs best on this problem, with the $\partial\mathbb{B}$ net second.

E.3. MNIST

The MNIST dataset (LeCun et al., 1998) consists of 60K training and 10K test examples of handwritten digits (0-9). We binarize the data by replacing pixels with grey value greater than 0.3 with 1, otherwise with 0. We initialize the network

	accuracy				
	mean	5 %ile	95 %ile	min	max
Tsetlin	99.3 +/- 0.3	95.9	100.0	91.6	100.0
$\partial\mathbb{B}$	**97.9 +/- 0.2**	**95.4**	**100.0**	**93.6**	**100.0**
neural network	95.4 +/- 0.5	90.1	98.6	88.2	99.9
SVM	58.0 +/- 0.3	56.4	59.2	55.4	66.5
naive Bayes	49.8 +/- 0.2	48.3	51.0	41.3	52.7
logistic regression	49.8 +/- 0.3	47.8	51.1	41.1	53.1

Table 3. Noisy XOR results measured over 100 experiments.

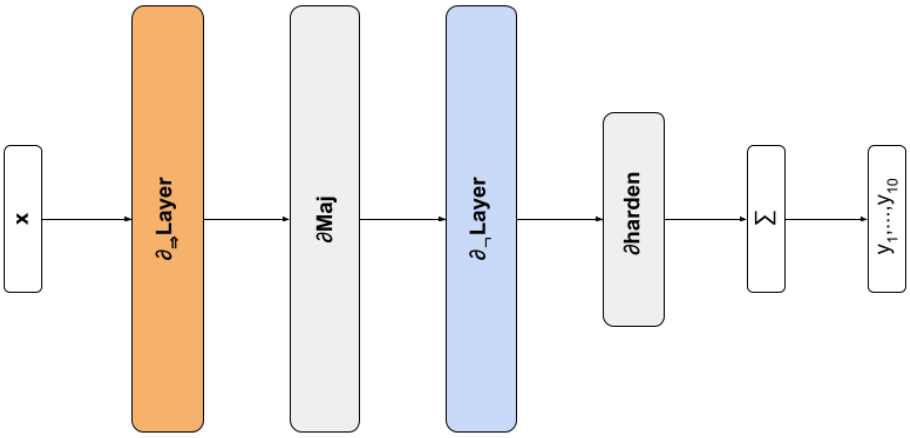

Figure 8. A non-convolutional $\partial\mathbb{B}$ net for MNIST. The input is a 28×28 bit matrix representing an image. The net consists of a $\partial_{\Rightarrow}\mathrm{Layer}$ (of width 60, to produce a 2940×16 reshaped array), a $\partial\mathrm{Maj}$ layer (to produce a vector of size 2940), a $\partial_{\neg}\mathrm{Layer}$ (of width 20, to produce a 20×2940 array), and a final $\partial\mathrm{harden}$ operator to generate hard-bits split into 10 buckets and summed to produce 10 integer logits. The net's weights, once hardened, consume 13.23 kb.

described in Figure 8 with random weights distributed as $w_i = 0.501 \times b + 0.3 \times (1 - b)$ where $b \sim \mathrm{Bernoulli}(0.01)$. We train for 1000 epochs with a batch size of 6000 using the RAdam optimizer and softmax cross-entropy loss.

We measure the accuracy on the final net. Table 4 compares the $\partial\mathbb{B}$ net against other classifiers (reference data taken from Granmo (2018)). Basic versions of the algorithms (e.g. no convolutional nets) are applied to unenhanced data (e.g. no data augmentation). The aim is to compare raw performance rather than optimise for MNIST. A 2-layer neural network trained on grey-value pixel data performs best. A Tsetlin machine of 40,000 automata each with 256 states (and therefore 40 kb of parameters) trained on binary data achieves $\approx 98.2\%$ accuracy. A $\partial\mathbb{B}$ net with 105,840 soft-bit weights that harden to 1-bit booleans (and therefore 13.23 kb of parameters) trained on binary data achieves $\approx 94.0\%$ accuracy. However, this $\partial\mathbb{B}$ net underfits the training data and we expect better performance from a larger model.

F. Proofs

Proposition F.1. $\partial_{\neg}(x, y) \blacktriangleright \neg(x \oplus y)$.

Proof. Table 5 is the truth table of the boolean function $\neg(x \oplus w)$, where $h(x) = \mathrm{harden}(x)$. □

Lemma F.2. *If a representative bit, x_i, is hard-equivalent to a target function, g, then so is the augmented bit, z.*

Proof. As x_i is representative then $\mathrm{harden}(x_i) = g(\mathrm{harden}(\mathbf{x}))$. The augmented bit, z, is given by (1):

$$z = \begin{cases} 1/2 + \bar{\mathbf{x}} \times |x_i - 1/2| & \text{if } x_i > 1/2 \\ x_i + \bar{\mathbf{x}} \times |x_i - 1/2| & \text{otherwise.} \end{cases}$$

	accuracy
2-layer NN, 800 HU, cross-entropy loss	98.6
Tsetlin	98.2 +/- 0.0
K-nearest-neighbours, L3	97.2
$\partial\mathbb{B}$	**94.0**
Logistic regression	91.5
Linear classifier (1-layer NN)	88.0
Decision tree	87.8
Multinomial Naive Bayes	83.2

Table 4. MNIST results. A classifier in *italics* was trained on grey-value pixel data, otherwise the classifier was trained on binarized data. Note: the $\partial\mathbb{B}$ results are from a small model that under-fits the data (due to OOM errors on my GPU). The next draft will include results using a larger $\partial\mathbb{B}$ net.

x	y	$h(x)$	$h(y)$	$\partial_\neg(x,y)$	$h(\partial_\neg(x,y))$	$\neg(h(y) \oplus h(x))$
$[0, \frac{1}{2})$	$[0, \frac{1}{2})$	0	0	$(\frac{1}{2}, 1]$	1	1
$(\frac{1}{2}, 1]$	$[0, \frac{1}{2})$	1	0	$[0, \frac{1}{2})$	0	0
$[0, \frac{1}{2})$	$(\frac{1}{2}, 1]$	0	1	$[0, \frac{1}{2})$	0	0
$(\frac{1}{2}, 1]$	$(\frac{1}{2}, 1]$	1	1	$(\frac{1}{2}, 1]$	1	1

Table 5. $\partial_\neg(x,y) \blacktriangleright \neg(y \oplus x)$.

In consequence,

$$\text{harden}(z) = \begin{cases} 1 & \text{if } x > 1/2 \\ 0 & \text{otherwise,} \end{cases}$$

since $x_i > 1/2 \Rightarrow z > 1/2$ and $x_i \leq 1/2 \Rightarrow z \leq 1/2$. Hence, $\text{harden}(z) = \text{harden}(x_i) = g(\text{harden}(\mathbf{x}))$ □

Proposition F.3. $\partial_\wedge(x,y) \blacktriangleright x \wedge y$.

Proof. Table 6 is the truth table of the boolean function $x \wedge y$, where $h(x) = \text{harden}(x)$.. □

x	y	$h(x)$	$h(y)$	$\partial_\wedge(x,y)$	$h(\partial_\wedge(x,y))$	$h(x) \wedge h(y)$
$[0, \frac{1}{2})$	$[0, \frac{1}{2})$	0	0	$[0, \frac{1}{2})$	0	0
$(\frac{1}{2}, 1]$	$[0, \frac{1}{2})$	1	0	$(\frac{1}{4}, \frac{1}{2})$	0	0
$[0, \frac{1}{2})$	$(\frac{1}{2}, 1]$	0	1	$(\frac{1}{4}, \frac{1}{2})$	0	0
$(\frac{1}{2}, 1]$	$(\frac{1}{2}, 1]$	1	1	$(\frac{1}{2}, 1]$	1	1

Table 6. $\partial_\wedge(x,y) \blacktriangleright x \wedge y$.

Proposition F.4. $\partial_\vee(x,y) \blacktriangleright x \vee y$.

Proof. Table 7 is the truth table of the boolean function $x \vee y$, where $h(x) = \text{harden}(x)$.. □

Proposition F.5. $\partial_\Rightarrow(x,y) \blacktriangleright x \Rightarrow y$.

Proof. Table 8 is the truth table of the boolean function $x \Rightarrow y$, where $h(x) = \text{harden}(x)$.. □

Lemma F.6. *Let* i = $\text{majority-index}(\mathbf{x})$*, then the* i*th element of* $\text{sort}(\mathbf{x})$ *is hard-equivalent to boolean majority, i.e.* $\text{harden}(\text{sort}(\mathbf{x})[i]) = \text{Maj}(\text{harden}(\mathbf{x}))$.

x	y	$h(x)$	$h(y)$	$\partial_\vee(x,y)$	$h(\partial_\vee(x,y))$	$h(x) \vee h(y)$
$[0, \frac{1}{2})$	$[0, \frac{1}{2})$	0	0	$[0, \frac{1}{2})$	0	0
$(\frac{1}{2}, 1]$	$[0, \frac{1}{2})$	1	0	$(\frac{1}{2}, 1]$	1	1
$[0, \frac{1}{2})$	$(\frac{1}{2}, 1]$	0	1	$(\frac{1}{2}, 1]$	1	1
$(\frac{1}{2}, 1]$	$(\frac{1}{2}, 1]$	1	1	$(\frac{1}{2}, 1]$	1	1

Table 7. $\partial_\vee(x,y) \blacktriangleright x \vee y$.

x	y	$h(x)$	$h(y)$	$\partial_\Rightarrow(x,y)$	$h(\partial_\Rightarrow(x,y))$	$h(x) \Rightarrow h(y)$
$[0, \frac{1}{2})$	$[0, \frac{1}{2})$	0	0	$(\frac{1}{2}, 1]$	1	0
$(\frac{1}{2}, 1]$	$[0, \frac{1}{2})$	1	0	$[0, \frac{1}{2})$	0	0
$[0, \frac{1}{2})$	$(\frac{1}{2}, 1]$	0	1	$(\frac{1}{2}, 1]$	1	0
$(\frac{1}{2}, 1]$	$(\frac{1}{2}, 1]$	1	1	$(\frac{1}{2}, \frac{7}{8})$	1	0

Table 8. $\partial_\Rightarrow(x,y) \blacktriangleright x \Rightarrow y$.

Proof. Let h denote the number of bits that are high in $\mathbf{x} = [x_1, \ldots, x_n]$. Then indices $\{j : n - h + 1 \leq j \leq n\}$ are high in $\mathrm{sort}(\mathbf{x})$. If the majority of bits are high, $h \geq \lfloor n/2 + 1 \rfloor$, then index $j = n - \lfloor n/2 + 1 \rfloor + 1 = n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ is high in $\mathrm{sort}(\mathbf{x})$. $\mathrm{majority\text{-}index}$ selects index $i = \lceil n/2 \rceil$ and therefore $i = j$. Hence, if the majority of bits are high then $\mathrm{sort}(\mathbf{x})[i]$ is high. Similarly, if the majority of bits are low, $h < \lfloor n/2 + 1 \rfloor$, then index $j = n - \lfloor n/2 + 1 \rfloor + 1 = n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ is low in $\mathrm{sort}(\mathbf{x})$. Hence, if the majority of bits are low then $\mathrm{sort}(\mathbf{x})[i]$ is low.

Note that $h \geq \lfloor n/2 + 1 \rfloor$ implies that $\mathrm{Maj}(\mathrm{harden}(\mathbf{x})) \geq \lfloor \frac{1}{2} + \frac{1}{n}\left(\frac{n}{2} + 1 - \frac{1}{2}\right) \rfloor \geq \lfloor 1 + \frac{1}{2n} \rfloor = 1$, and $h < \lfloor n/2 + 1 \rfloor$ implies that $\mathrm{Maj}(\mathrm{harden}(\mathbf{x})) < \lfloor 1 + \frac{1}{2n} \rfloor = 0$.

In consequence, $\mathrm{harden}(\mathrm{sort}(\mathbf{x})[i]) = \mathrm{Maj}(\mathrm{harden}(\mathbf{x}))$ for all $h \in [0, \ldots, n]$. □

Theorem F.7. $\partial\mathrm{Maj} \blacktriangleright \mathrm{Maj}$.

Proof. $\partial\mathrm{Maj}$ augments the representative bit $x_i = \mathrm{sort}(\mathbf{x})[\mathrm{majority\text{-}index}(\mathbf{x})]$. By lemma F.6 the representative bit is $\blacktriangleright\ \mathrm{Maj}(\mathrm{harden}(\mathbf{x}))$. By lemma F.2, the augmented bit, $\mathrm{augmented\text{-}bit}(\mathrm{sort}(\mathbf{x}), \mathrm{majority\text{-}index}(\mathbf{x}))$, is also $\blacktriangleright \mathrm{Maj}(\mathrm{harden}(\mathbf{x}))$. Hence $\partial\mathrm{Maj} \blacktriangleright \mathrm{Maj}$. □

Proposition F.8. $\partial\mathrm{count\text{-}hot} \blacktriangleright \mathrm{count\text{-}hot}$.

Proof. Let l denote the number of bits that are low in $\mathbf{x} = [x_1, \ldots, x_n]$, and let $\mathbf{y} = \partial\mathrm{count\text{-}hot}(\mathbf{x})$. Then $\mathbf{y}[l+1]$ is high and any $\mathbf{y}[i]$, where $i \neq l + 1$, is low. Let $\mathbf{z} = \mathrm{count\text{-}hot}(\mathrm{harden}(\mathbf{x}))$. Then $\mathbf{z}[l+1]$ is high and any $\mathbf{z}[i]$, where $i \neq l + 1$, is low. Hence, $\mathrm{harden}(\mathbf{y}) = \mathbf{z}$, and therefore $\partial\mathrm{count\text{-}hot} \blacktriangleright \mathrm{count\text{-}hot}$. □