$\partial\mathbb{B}$ NETS: LEARNING DISCRETE FUNCTIONS BY GRADIENT DESCENT

Ian Wright*

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. 'Hardening' involves no loss of accuracy, unlike existing approaches to neural network binarization. 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 are differentiable functions with weights represented by machine floats. Networks are trained by gradient descent in weight-space, where the direction of descent minimises loss. The gradients are efficiently calculated by the backpropagation algorithm (Rumelhart et al., 1986). This overall approach has led to tremendous advances in machine learning.

However, there are drawbacks. First, differentiability means we cannot directly learn discrete functions, such as logical predicates. In consequence, what a network has learned is difficult to interpret and verify. Second, representing weights as machine floats enables time-efficient training but at the cost of memory-inefficient models. For example, network quantisation techniques (see Qin et al. (2020)) demonstrate that full 64 of 32-bit precision weights are often unnecessary for final predictive performance, although there is a trade-off.

A standard approach to mitigate these drawbacks is to 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, we define $\partial\mathbb{B}$ nets that have two equivalent aspects: a *soft-net*, which is a differentiable real-valued function, and a *hard-net*, which is a non-differentiable, discrete function. Both aspects are semantically equivalent. We train the soft-net as normal, using backpropagation, then 'harden' the learned weights to boolean values, which we then bind with the hard-net to yield a discrete function with identical predictive performance (see figure 1). In consequence, interpreting and verifying a $\partial\mathbb{B}$ net is relatively less difficult. And boolean-valued, 1-bit weights significantly increase the memory-efficiency of trained models.

The main contributions of this work are (i) defining novel activation functions that 'harden' to semantically equivalent discrete functions, (ii) defining novel network architectures to effectively learn discrete functions that solve multi-class classification problems, and (iii) experiments that demonstrate $\partial\mathbb{B}$ nets compete with existing approaches in terms of predictive performance yet yield compact models.

Section 2 discusses related work, section 3 defines $\partial\mathbb{B}$ nets, section 4 presents experimental results, and section 5 concludes.

*wrighti@acm.org

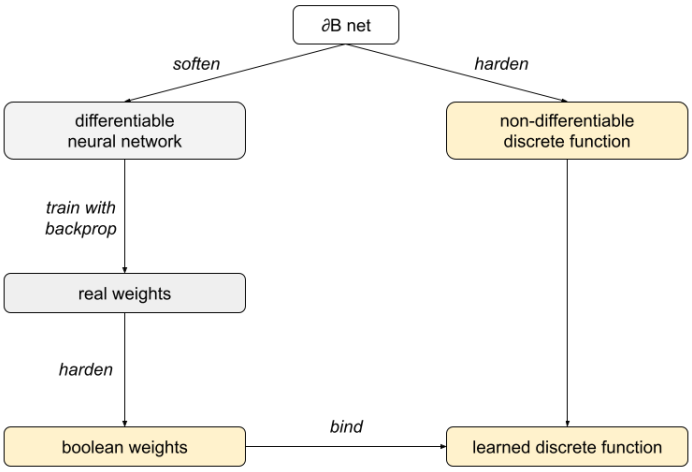

Figure 1: *Learning discrete functions with a $\partial\mathbb{B}$ net.* A $\partial\mathbb{B}$ net specifies (i) a differentiable neural network that is hard-equivalent to (ii) a non-differentiable discrete function. The neural network is trained as normal with backpropagation to yield a set of real weights. The real weights are hardened to boolean values and then bound with the discrete function. The result is a learned discrete function that performs identically to the trained network.

2 Related work

Methods that learn discrete boolean functions can be broadly categorized as either non-differentiable or differentiable.

Non-differentiable approaches include boolean-valued decision trees (Breiman et al., 1984), random forests (Ho, 1995), genetic programming (Koza, 1992) and, more recently, Tsetlin machines Granmo (2018). Tsetlin machines represent propositional formulae by collections of simple automata with integer weights optimised by positive and negative feedback defined in terms of a hard threshold function. These models directly represent boolean decisions and therefore are easier to interpret compared to deep neural networks. However, they tend to perform less well, compared to differentiable approaches such as deep learning, on large volumes of high-dimensional data (e.g. NLP, images and audio) without manual feature engineering, although Tsetlin machines show promise on such tasks (Granmo et al., 2019).

Differentiable approaches that learn boolean functions include (i) systems that integrate rule-based reasoning with neural components, and (ii) binarization techniques that quantize neural networks by converting real-valued weights and activations to binary values. For example, differentiable inductive logic programming (Evans & Grefenstette, 2018), neural logic machines (Dong et al., 2019) and differentiable neural logic networks Payani (2020) learn first-order logic rules using gradient descent. These systems combine the benefits of logical inference and interpretability with end-to-end differentiability. However, they rely on combinatoric enumeration of rulesets and therefore do not scale to large datasets. The technique of network binarization aims to significantly reduce model size and inference costs while maintaining predictive accuracy. Binarization reduces a real-valued neural network to a binary network where nonlinear activation functions are replaced by boolean majority functions. For example, BinaryConnect (Courbariaux et al., 2015), XNOR-Net (Rastegari et al., 2016), and LUTNet (Wang et al., 2020), optimize a continuous relaxation or approximation of the binary net during training. However, binary-valued functions are intrinsically non-differentiable and therefore training by gradient descent is challenging. Plus, binarization throws away information, which reduces accuracy (Qin et al., 2020).

The design-space of algorithms that learn boolean functions is large, with various trade-offs. In this paper we investigate an under-explored area of differentiable nets that are semantically equivalent, without approximation or loss, to an arbitrarily complex boolean function. We aim to combine the

benefits of deep neural networks trained by gradient descent with the efficiency, interpretability and logical bias of boolean functions – but without loss of accuracy.

3 $\partial\mathbb{B}$ NETS

A $\partial\mathbb{B}$ net has two aspects, a soft-net and a hard-net. Both nets use bits to represent transitory values and learnable weights, but a soft-net uses soft-bits and a hard-net uses hard-bits.

Definition (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 (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*

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

The soft-bit value $1/2$ is therefore a threshold. Above this threshold the soft-bit represents True, otherwise it represents False.

A soft-net is any differentiable function, f, that 'hardens' to a semantically equivalent discrete function, g. For example, if $f(x) = 1 - x$, where $x \in [0, 1]$, and $g(y) = \neg y$, where $y \in \{0, 1\}$ then: if x is high (resp. low) then both $f(x)$ and $g(\text{harden}(x))$ are low (resp. high). In other words, f is hard-equivalent to boolean negation. More generally:

Definition (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* $f \blacktriangleright g$*.*

Neural networks are typically composed of nonlinear activation functions (for representational generality) that are strictly monotonic (so gradients always exist that link changes in inputs to outputs without local minima) and smooth (so gradients reliably represent the local loss surface). However, activation functions that are monotonic but not strictly (so some gradients are zero) and differentiable almost everywhere (so some gradients are undefined) can also work, e.g. RELU (Nair & Hinton, 2010). $\partial\mathbb{B}$ nets are composed from 'activation' functions that also satisfy these properties plus the additional property of hard-equivalence to a boolean function (and natural generalisations). We now turn to specifying the kind of 'activation' functions used by $\partial\mathbb{B}$ nets.

3.1 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 negate. The boolean function that meets this requirement is $\neg(x \oplus w)$. However, this function is not differentiable. Define the differentiable function,

$$\begin{aligned} \partial_\neg : [0, 1]^2 &\to [0, 1], \\ (w, x) &\mapsto 1 - w + x(2w - 1), \end{aligned}$$

where $\partial_\neg(w, x) \blacktriangleright \neg(x \oplus w)$ (see proposition 1).

There are many kinds of differentiable fuzzy logic operators (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 2).

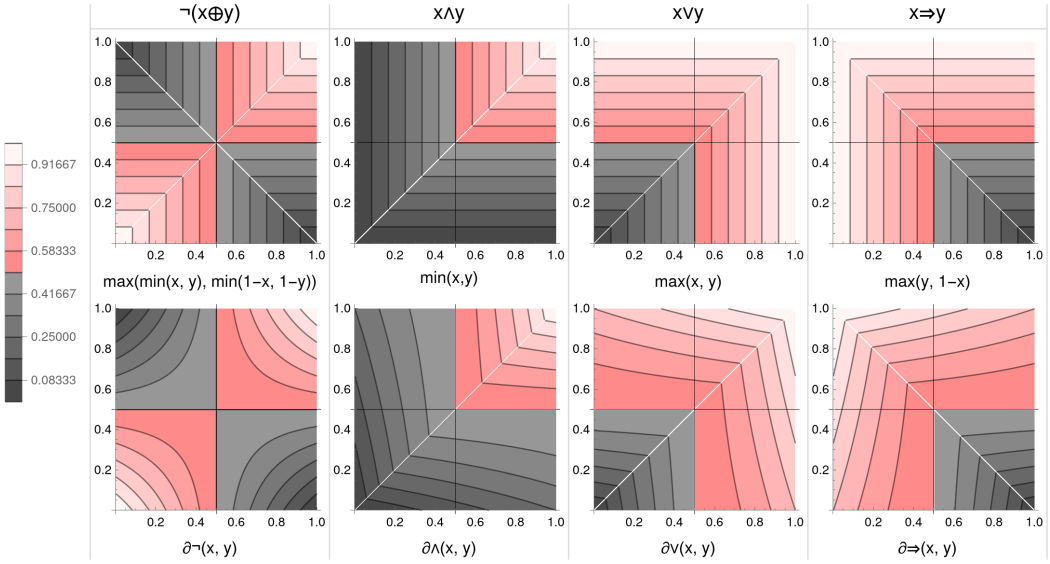

Figure 2: *Gradient-rich versus gradient-sparse differentiable boolean functions.* Each column contains contour plots of functions $f(x, y)$ that are hard-equivalent to a boolean function (one of $\neg(x \oplus y)$, $x \wedge y$, $x \vee y$, or $x \Rightarrow y$). Every function is continuous and differentiable almost everywhere (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.

Definition (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 must be composed of 'activation' functions that are hard-equivalent to discrete functions but also, where possible, gradient-rich. To meet this requirement we introduce the technique of margin packing.

3.2 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 we have a vector of soft-bit inputs $\mathbf{x}$ and the ith element 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

$$\begin{aligned} \text{margin-fraction} : [0, 1]^n \times 1, 2, \ldots, n &\to [0, 1], \\ (\mathbf{x}, i) &\mapsto \bar{\mathbf{x}} \times |x_i - 1/2| \,. \end{aligned}$$

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

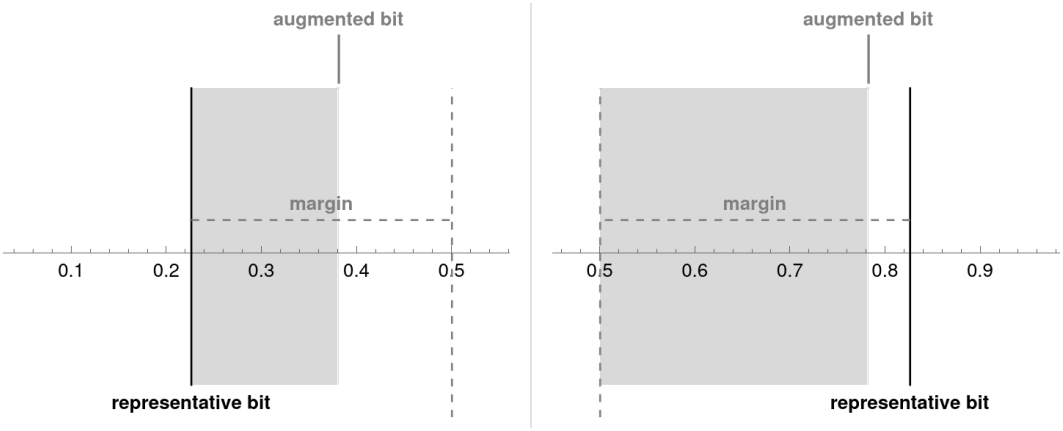

Figure 3: *Margin packing for constructing gradient-rich, hard-equivalent functions*. A representative bit, z, is hard-equivalent to a discrete target function but gradient-sparse (e.g. $z = \min(x, y) \blacktriangleright x \wedge y$). On the left z is low, $z < 1/2$; on the right z is high, $z > 1/2$. 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 $\approx 90\%$.

$1/2$. At the threshold point there is no margin to pack. Now, define the augmented bit as

$$\text{augmented-bit} : [0,1]^n \times 1, 2, \ldots, n \to [0,1],$$
$$(\mathbf{x}, i) \mapsto \begin{cases} 1/2 + \text{margin-fraction}(\mathbf{x}, i) & \text{if } x_i > 1/2 \\ x_i + \text{margin-fraction}(\mathbf{x}, i) & \text{otherwise.} \end{cases} \quad (1)$$

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 3). Note that if the ith bit is representative (i.e. hard-equivalent to the target function) then so is the augmented bit (see lemma 1). We use margin packing, where appropriate, to define gradient-rich, hard-equivalents of boolean functions.

3.3 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

$$\partial_\wedge : [0,1]^n \to [0,1],$$
$$\mathbf{x} \mapsto \text{augmented-bit}(\mathbf{x}, \text{argmin}_i\, x[i])$$

is therefore hard-equivalent to the boolean function $\bigwedge_{i=1}^n x_i$ (see proposition 2). In the special case $n = 2$ we get the piecewise function,

$$\partial_\wedge(x, y) = \begin{cases} 1/2 + 1/2(x + y)(\min(x, y) - 1/2) & \text{if } \min(x, y) > 1/2 \\ \min(x, y) + 1/2(x + y)(1/2 - \min(x, y)) & \text{otherwise.} \end{cases}$$

Note that $\partial_\wedge$ is differentiable almost everywhere and gradient-rich (see figure 2).

The differentiable analogue of $\vee$ is identical to $\wedge$, except the representative bit is selected by $\max$. The function

$$\partial_\vee : [0,1]^n \to [0,1],$$
$$\mathbf{x} \mapsto \text{augmented-bit}(\mathbf{x}, \text{argmax}_i\, x[i])$$

is hard-equivalent to the boolean function $\bigvee_{i=1}^n x_i$ (see proposition 3). Note that $\partial_\vee$ is differentiable almost everywhere and gradient-rich (see figure 2).

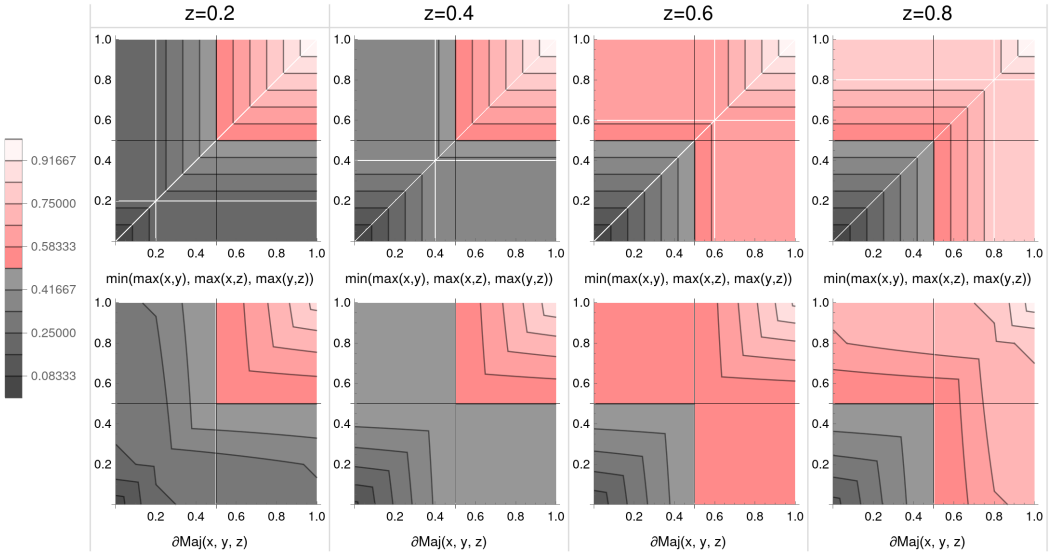

Figure 4: *Differentiable boolean majority.* The boolean majority function for three variables in DNF form is $\mathrm{Maj}(x, y, z) = (x \wedge y) \vee (x \wedge y) \vee (y \wedge z)$. The upper row contains contour plots of $f(x, y, z) = \min(\max(x, y), \max(x, z), \max(y, z))$ for values of $z \in \{0.2, 0.4, 0.6, 0.8\}$. 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.

The differentiable analogue of $\Rightarrow$ (material implication) is defined in terms of $\partial_\vee$. The function

$$\begin{aligned} \partial_\Rightarrow : [0, 1]^2 &\to [0, 1], \\ (x, y) &\mapsto \partial_\vee(y, 1 - x), \end{aligned}$$

is hard-equivalent to $x \Rightarrow y$ (see proposition 4). We can define analogues of all the basic boolean operators in a similar manner.

3.4 Differentiable majority

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

$$\begin{aligned} \mathrm{Maj} : \{0, 1\}^n &\to \{0, 1\}, \\ \mathbf{x} &\mapsto \left\lfloor \frac{1}{2} + \frac{\sum_{i=1}^n x_i - 1/2}{n} \right\rfloor, \end{aligned}$$

where we count False as 0 and True as 1. Interpret each input bit x_i as a vote, yes or no, for a binary decision. If the majority of voters are in favour then Maj outputs 1. The majority function, in the context of a predictive model, aggregates multiple bits of weak evidence into a hard decision.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 $k = \lceil n/2 \rceil$. Each clause checks whether a unique combination of a majority of the n bits are all high, e.g. $\mathrm{Maj}(x, y, z) = (x \wedge y) \vee (x \wedge y) \vee (y \wedge z)$. 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 (e.g. $n = 50$ generates over 100 trillion clauses, which is infeasible). And no general algorithm exists to find the minimal representation of Maj for arbitrary n.

Instead, we trade-off time for memory costs. Observe that if the function $\mathrm{sort}(\mathbf{x})$ sorts the elements of $\mathbf{x}$ in ascending order then the 'median' soft-bit is representative. For example, if

$\mathbf{x} = [0.4, 0.9, 0.2]$ then $\mathrm{sort}(\mathbf{x}) = [0.2, 0.4, 0.9]$ and the 'median' bit $x_2 = 0.4$ is low, which is hard-equivalent to $\mathrm{Maj}(0, 1, 0) = 0$. Define the index of the 'median' bit by

$$\begin{aligned} \text{majority-index} : [0,1]^n &\to \mathbb{Z}_{>0} \\ \mathbf{x} &\mapsto \left\lceil \frac{|\mathbf{x}|}{2} \right\rceil. \end{aligned}$$

Then, applying margin packing, define the differentiable function

$$\begin{aligned} \partial\mathrm{Maj} : [0,1]^n &\to [0,1], \\ \mathbf{x} &\mapsto \text{augmented-bit}(\mathrm{sort}(\mathbf{x}), \text{majority-index}(\mathbf{x})), \end{aligned}$$

which is hard-equivalent to Maj (see theorem 1). Note that $\partial\mathrm{Maj}$ is differentiable almost everywhere and gradient-rich (see figure 4). If sort is quicksort then the the average time-complexity of $\partial\mathrm{Maj}$ is $\mathcal{O}(n \log n)$, which makes $\partial\mathrm{Maj}$ more expensive than $\partial_\neg$, $\partial_\wedge$, $\partial_\vee$ and $\partial_\Rightarrow$ at training time. 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 sorting 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)).

3.5 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 $\mathrm{count}(\mathbf{x}, k)$ where $c(\mathbf{x}) := |\{x_i : x_i = 1\}| = k$ (i.e. 'exactly k high'), which can be useful in multiclass classification problems.

As before, we use sort to trade-off time for memory costs. Observe that if the elements of $\mathbf{x}$ are in ascending order then, if any soft-bits are high, there exists a unique contiguous pair of indices $(i, i+1)$ where x_i is low and x_{i+1} is high, where index i is a direct count of the number of soft-bits that are low in $\mathbf{x}$. In consequence, define

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

where

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

$\partial\text{count-hot}(\mathbf{x})$ outputs a 1-hot vector where the index of high bit is the number of low bits in $\mathbf{x}$. For example, $\partial\text{count-hot}([0.1, 0.9, 0.2]) = [0.1, 0.2, \mathbf{0.8}, 0.1]$, indicating that 2 bits are low, and $\partial\text{count-hot}([0.6, 0.9, 0.7]) = [\mathbf{0.6}, 0.4, 0.3, 0.1]$, indicating that 0 bits are low. Note that $\partial\text{count-hot}$ is differentiable, gradient-rich and hard-equivalent to the boolean function

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

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 5). 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, $\partial\mathrm{count}(\mathbf{x}, k)$ is straightforwardly $\partial\text{count-hot}(\mathbf{x})[k]$ where we can use margin-packing to ensure that this single soft-bit is gradient-rich.

This basic set of boolean functions is sufficient to learn non-trivial relationships from data. We now turn to constructing $\partial\mathbb{B}$ nets from compositions of these functions.

3.6 BOOLEAN LOGIC LAYERS

The fully variety of $\partial\mathbb{B}$ net architectures is to be explored. Here we focus on defining basic layers sufficient for the classification experiments in section 4. Other kinds of layers, such as convolutional, or real encoders/decoders for regression problems, will be addressed in a sequel.

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

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

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:

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

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:

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

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.

3.7 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:

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

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.

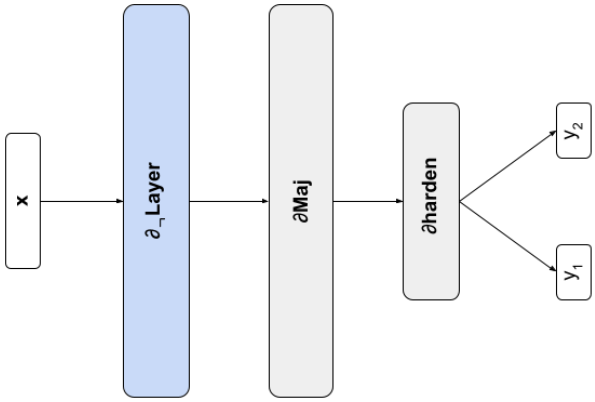

Figure 5: *A $\partial\mathbb{B}$ net to illustrate hardening*. The net concatenates a $\partial_{\neg}\mathrm{Layer}$ (of width n) with a reshaping layer that outputs two vectors, which get reduced, by a $\partial\mathrm{Maj}$ operator, to 2 soft-bits, one for each class label. A final $\partial\mathrm{harden}$ layer ensures the loss is a function of hard bits. When $|\mathbf{x}| = 1$ we choose $n = 2$ and therefore the net's weights, once hardened, consume 2 bits. When $|\mathbf{x}| = 5$ we choose $n = 8$ and the weights consume 40 bits (5 bytes).

$\partial\mathbb{B}$ 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):

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

where

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

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

4 Experiments

The $\partial\mathbb{B}$ 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 $\partial\mathbb{B}$ 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 $\partial\mathbb{B}$ nets against standard ML approaches on three problems: the classic Iris dataset, an adversarial noisy XOR problem, and MNIST. But first we illustrate the kind of discrete program that a $\partial\mathbb{B}$ net learns.

4.1 Hardening

We present a toy problem to illustrate hard-equivalence. Consider the trivial problem of predicting whether a person wears a $\mathrm{t\text{-}shirt}$ (label 0) or a coat (label 1) conditional on the single feature $\mathrm{outside}$ (0 = False, and 1 = True). The training and test data consist of the examples in table 1.

We use the $\partial\mathbb{B}$ net described in figure 5, which is hard-equivalent to the discrete program:

```
def dbNet(outside):
  return [
    ge(sum((0, not(xor(ne(outside, 0), w1)))), 1),
       ge(sum((0, not(xor(ne(outside, 0), w2)))), 1)
  ]
```

outside	label
0	0
1	1

Table 1: A trivial learning problem

with trainable weights w_1 and w_2. We randomly initialize the network and train using the RAdam optimizer (Liu et al., 2020) with softmax cross-entropy loss until training and test accuracies are both 100%. We harden the learned weights to get $w_1 = \text{False}$ and $w_2 = \text{True}$, and bind with the discrete program, which then symbolically simplifies to:

```
def dbNet(outside):
        return [not(outside), outside]
```

which is directly interpretable as 'when outside wear a coat, otherwise wear a t-shirt'.

Hardening scales to arbitrarily complex $\partial\mathbb{B}$ nets. Interpreting the net's predictions requires automatic symbolic simplification. For example, introduce 4 additional boolean features: very-cold, cold, warm, and very-warm. The training and test data consists of examples like those in table 2.

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 2: A toy learning problem

We use the same architecture but increase the width of the $\partial\neg\text{Layer}$ from 2 to 8. The net is now hard-equivalent to the discrete program:

```
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((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((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)
  ]
```

We train the $[w_1, \ldots, w_{40}]$ soft-bit weights as before then harden to 40 boolean weights and bind with the discrete program. Post-training 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
  ]
```

The predictions linearly weight multiple pieces of evidence due to the presence of the ∂Maj operator (which is probably overkill for this toy problem). From this expression 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. The discrete program is more interpretable compared to typical neural networks, and can be exactly encoded as a SAT problem in order to verify its properties, such as robustness.

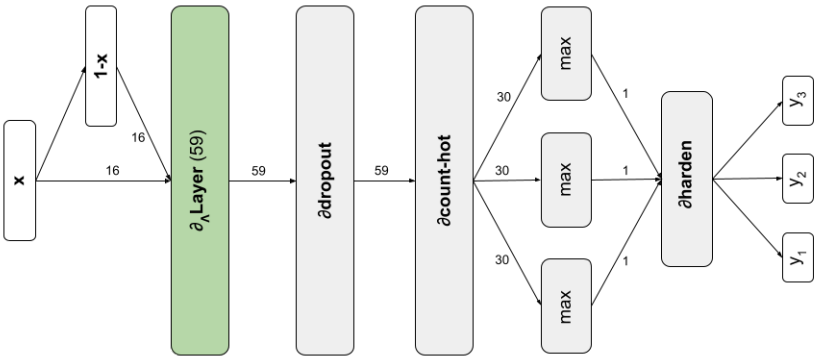

Figure 6: *A $\partial\mathbb{B}$ net for the binary Iris problem*. The net concatenates the soft-bit input, $\mathbf{x}$ (length 16), with its negation, $\mathbf{1} - \mathbf{x}$, and supplies the resulting vector (length 32) to a $\partial_\wedge\mathrm{Layer}$ (width 59), a $\partial\mathrm{dropout}$ layer for improved generalisation, a $\partial\mathrm{count\text{-}hot}$ 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.

	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 3: *Ranked binary Iris results* measured over 1000 experiments.

4.2 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 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 3 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.

4.3 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 $\partial_\wedge\mathrm{Layer}$, $w_i = 0.501 \times b + 0.3 \times (1 - b)$ where $b \sim \mathrm{Bernoulli}(0.01)$; in the $\partial_\vee\mathrm{Layer}$, $w_i = 0.7 \times b + 0.499 \times (1 - b)$ where $b \sim \mathrm{Bernoulli}(0.99)$); and in the $\partial_\neg\mathrm{Layer}$, $w_i \sim \mathrm{Uniform}(0.499, 0.501)$. 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 4 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 constrast, the

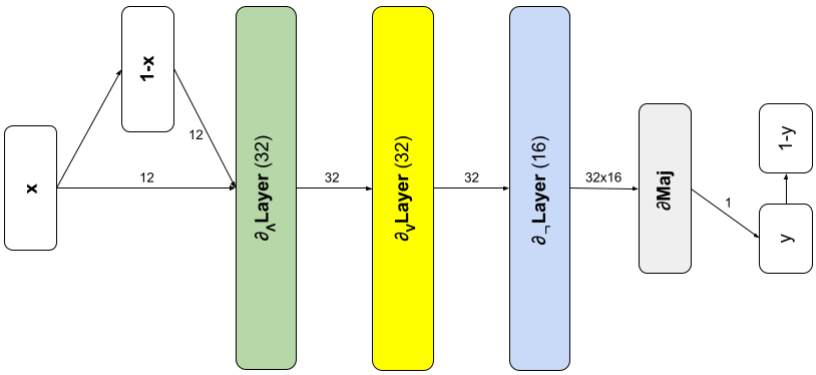

Figure 7: *A $\partial\mathbb{B}$ net for the noisy xor problem.* The net concatenates the soft-bit input, $\mathbf{x}$ (length 12), with its negation, $\mathbf{1} - \mathbf{x}$, and supplies the resulting vector (length 24) to a $\partial_{\wedge}\mathrm{Layer}$ (width 32), $\partial_{\vee}\mathrm{Layer}$ (width 32), $\partial_{\neg}\mathrm{Layer}$ (width 16), and a final $\partial\mathrm{Maj}$ 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.

	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 4: *Ranked noisy XOR results* measured over 100 experiments.

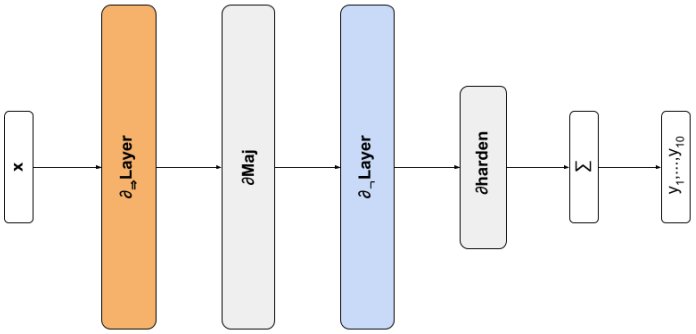

Figure 8: *A non-convolutional $\partial\mathbb{B}$ net for MNIST.* The input is a 28×28 bit matrix representing an image. The net consists of a $\partial_{\Rightarrow}\mathrm{Layer}$ (of width 60, to produce a 2940×16 reshaped array), a $\partial\mathrm{Maj}$ layer (to produce a vector of size 2940), 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.

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.

4.4 MNIST

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

	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 5: *Ranked 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 underfits the data (due to OOM errors on my GPU). The next draft will include results using a larger $\partial\mathbb{B}$ net.

1, otherwise with 0. We initialize the network described in figure 8 with random weights distributed as $w_i = 0.501 \times b + 0.3 \times (1 - b)$ where $b \sim \text{Bernoulli}(0.01)$. 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 5 compares the $\partial\mathbb{B}$ net against other classifiers (reference data taken from Granmo (2018) and `yann.lecun.com/exdb/mnist`). 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.

5 Conclusion

$\partial\mathbb{B}$ nets are differentiable neural networks that are hard-equivalent to non-differentiable, boolean-valued functions. $\partial\mathbb{B}$ nets can therefore learn discrete functions by gradient descent. The main novelty of $\partial\mathbb{B}$ nets is the semantic equivalence between their two aspects: a differentiable soft-net and a non-differentiable hard-net. Maintaining this semantic equivalence requires defining new kinds of differentiable functions that are hard-equivalent to boolean functions, such as non-differentiable boolean majority. We propose 'margin packing' as a potentially general technique for constructing differentiable functions that are hard-equivalent yet gradient-rich (and therefore backpropagate error to all their inputs). An advantage of $\partial\mathbb{B}$ nets is that we train the soft-net using efficient backpropagation on GPUs then 'harden' to generate a learned discrete function that, unlike existing approaches to neural network binarization, has provably identical accuracy.

$\partial\mathbb{B}$ nets, being ultimately of a discrete and logical nature, are easier to interpret compared to standard neural networks, for example generating propositional formulae that can be further analysed, either by symbolic simplification or verification by SAT solvers. These properties are important in safety-critical domains. In addition, $\partial\mathbb{B}$ nets at inference time are highly compact, due to 1-bit weights, and potentially cheap to evaluate, as they reduce to bit manipulation and integer arithmetic. These properties are important in resource-poor deployment environments, such as edge devices. Further, due to the differentiable nature of $\partial\mathbb{B}$ nets, they can be arbitrarily composed with standard neural nets (e.g. by embedding them within standard nets to introduce domain-specific logical bias).

Preliminary experiments on three classification benchmarks demonstrate that $\partial\mathbb{B}$ nets can outperform multilayer perceptron networks, support vector machines, decision trees, and logistic regression. In terms of classification accuracy, the non-differentiable Tsetlin machine outperforms $\partial\mathbb{B}$ nets, which indicates room for futher improvements, e.g. by defining more expressive $\partial\mathbb{B}$ net layers (threshold functions with a learnable integer threshold, boolean decision lists etc.) and architectures (convolutional, regression nets, skip connections, attention etc.). In other words, this paper is only a first step towards exploring the space of differentiable nets that satisfy the requirement of hard-equivalence.

ACKNOWLEDGMENTS

Thanks to GitHub Next for sponsoring this research. And thanks to Pavel Augustinov, Richard Evans, Johan Rosenkilde, Max Schaefer, Ganesh Sittampalam, Tamás Szabó and Albert Ziegler for helpful discussions and feedback.

REFERENCES

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

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Leo Breiman, Jerome Friedman, Charles J. Stone, , and R.A. Olshen. *Classification and Regression Trees*. Chapman and Hall/CRC, 1984.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 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.

Marco Cuturi, Olivier Teboul, and Jean-Philippe Vert. Differentiable ranking and sorting using optimal transport. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (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.

Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. Neural logic machines. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=B1xY-hRctX.

Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *J. Artif. Int. Res.*, 61(1):1–64, jan 2018. ISSN 1076-9757.

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

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

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

Ole-Christoffer Granmo, Sondre Glimsdal, Lei Jiao, Morten Goodwin Olsen, Christian Walter Peter Omlin, and Geir Thore Berge. The convolutional tsetlin machine. *ArXiv*, abs/1905.09688, 2019.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2023. URL http://github.com/google/flax.

Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pp. 278–282 vol.1, 1995. doi: 10.1109/ICDAR.1995.598994.

J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. A Bradford book. Bradford, 1992. ISBN 9780262111706. URL https://books.google.co.uk/books?id=Bhtxo60BV0EC.

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

Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rkgz2aEKDr.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pp. 807–814, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.

Ali Payani. *Differentiable neural logic networks and their application onto inductive logic programming*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2020. URL https://hdl.handle.net/1853/62833.

Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. 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.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (eds.), *Computer Vision – ECCV 2016*, pp. 525–542, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46493-0.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 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.

Emile van Krieken, Erman Acar, and Frank van Harmelen. 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.

E. Wang, J. J. Davis, P. K. Cheung, and G. A. Constantinides. Lutnet: Learning fpga configurations for highly efficient neural network inference. *IEEE Transactions on Computers*, 69(12):1795–1808, dec 2020. ISSN 1557-9956. doi: 10.1109/TC.2020.2978817.

APPENDIX

A PROOFS

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

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

Lemma 1. *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 $\text{harden}(x_i) = g(\text{harden}(\mathbf{x}))$. The augmented bit, z, is given by equation 1:

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

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 6: $\partial_{\neg}(x,y) \blacktriangleright \neg(y \oplus x)$.

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 2. $\partial_{\wedge}(x,y) \blacktriangleright x \wedge y$.

Proof. Table 7 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 7: $\partial_{\wedge}(x,y) \blacktriangleright x \wedge y$.

Proposition 3. $\partial_{\vee}(x,y) \blacktriangleright x \vee y$.

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

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 8: $\partial_{\vee}(x,y) \blacktriangleright x \vee y$.

Proposition 4. $\partial_{\Rightarrow}(x,y) \blacktriangleright x \Rightarrow y$.

Proof. Table 9 is the truth table of the boolean function $x \Rightarrow y$, where $h(x) = \text{harden}(x)$.. □

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

Table 9: $\partial_{\Rightarrow}(x,y) \blacktriangleright x \Rightarrow y$.

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

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

Theorem 1. $\partial\mathrm{Maj} \blacktriangleright \mathrm{Maj}$.

Proof. $\partial\mathrm{Maj}$ augments the representative bit $x_i = \mathrm{sort}(\mathbf{x})[\mathrm{majority\text{-}index}(\mathbf{x})]$. By lemma 2 the representative bit is $\blacktriangleright\ \mathrm{Maj}(\mathrm{harden}(\mathbf{x}))$. By lemma 1, 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 5. $\partial\mathrm{count\text{-}hot} \blacktriangleright \mathrm{count\text{-}hot}$.

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 $\partial\mathrm{count\text{-}hot} \blacktriangleright \mathrm{count\text{-}hot}$. □