Reverse Derivative Ascent:
A Categorical Approach to Learning Boolean Circuits

Paul Wilson
University College London
University of Southampton
paul@statusfailed.com

Fabio Zanasi
University College London
f.zanasi@ucl.ac.uk

We introduce *Reverse Derivative Ascent*: a categorical analogue of gradient based methods for machine learning. Our algorithm is defined at the level of so-called *reverse differential categories*. It can be used to learn the parameters of models which are expressed as morphisms of such categories. Our motivating example is boolean circuits: we show how our algorithm can be applied to such circuits by using the theory of reverse differential categories. Note our methodology allows us to learn the parameters of boolean circuits *directly*, in contrast to existing binarised neural network approaches. Moreover, we demonstrate its empirical value by giving experimental results on benchmark machine learning datasets.

1 Introduction

Computation of the reverse derivative is a critical part of gradient-based machine learning methods (see e.g. [16] for an overview). In essence, the reverse derivative tells us how to update the parameters of a model, given a prediction error. This update procedure is at the core of many optimisation methods, such as *stochastic gradient descent* [16, 2.2] –used for training deep neural networks.

Now, the model class of choice is typically neural networks, which can be considered as the smooth maps $\mathbb{R}^a \to \mathbb{R}^b$. A natural question to ask is whether these gradient based methods could be generalised to other settings. In this paper, we focus on boolean functions - maps $\mathbb{Z}_2^a \to \mathbb{Z}_2^b$, and their operational counterpart, boolean circuits.

This setting has real practical value. Larger neural network models typically require expensive and power-hungry GPGPU hardware to train and run [4] [14]. Performance can be improved via *binarisation*: the extraction from a trained neural network of a boolean circuit, which provides a better optimised representation of the same model.

The usual pattern in these approaches (see e.g. BinaryConnect [4] and LUTNet [20]) is to perform the training aspect exclusively on the 'real-valued' side. However, training schemes for binarised models such as boolean circuits are typically more efficient [9]. It is thus natural to ask: "why not learn the parameters of boolean circuits directly?"

Reverse Derivative Ascent, the algorithm that we introduce in this paper, originates from this question: instead of training a neural network and then extracting a boolean circuit, we begin with a boolean circuit, and learn its parameters directly.

In defining and analysing the algorithm, we take a categorical approach. Our methodology relies on the abstract framework provided by *reverse differential categories* [3], which axiomatises the concept of reverse derivative operator. We proceed in three steps:

1. We give a syntactic presentation in terms of string diagrams for the reverse differential operator of polynomials, and a safety condition specifying when we can apply this operator to boolean circuits.

David I. Spivak and Jamie Vicary (Eds.): Applied Category Theory 2020 (ACT2020)
EPTCS ??, 2020, pp. 1–14, doi:10.4204/EPTCS.??.??

2. We define Reverse Derivative Ascent as ‘gradient’-based algorithm working for arbitrary morphisms of reverse differential categories.
3. We can then apply Reverse Derivative Ascent to boolean circuits, and demonstrate its empirical value by giving experimental results for benchmark datasets.

The categorical setting brings two main advantages. First, by exploiting the presentation of boolean circuits as an axiomatic theory of string diagrams [12], we are able to define a suitable reverse derivative operator *compositionally*, by induction on the circuit syntax. Second, because our definition of Reverse Derivative Ascent is phrased at the general level of reverse differential categories, it paves the way for the application to model classes other than boolean circuits, which we leave for future work.

The rest of the paper is structured as follows. We begin with necessary background in Section 2, defining (parametrised) boolean functions and circuits. In Section 3 we define our graphical operator for boolean circuits, give a safety condition for its application, and show that it is consistent with the reverse differential combinator of polynomials. We include additional material relevant to Section 3 in Appendix A. In Section 4, we describe the Reverse Derivative Ascent algorithm and provide a Haskell library to run our algorithm on boolean circuits[1]. Also, we give empirical evidence that it is able to learn functions from data. We conclude the paper with a discussion of future work in Section 5.

2 Background: Boolean Functions, Circuits, and Polynomials

We first recall the basics of boolean functions.

Definition 1. A *boolean function* is a map $f : \mathbb{Z}_2^a \to \mathbb{Z}_2^b$. We also say that a map $g : \mathbb{Z}_2^{p+a} \to \mathbb{Z}_2^b$ is a *parametrised boolean function* with p parameters, a inputs, and b outputs. We denote by **BoolFun** the symmetric strict monoidal category whose objects are natural numbers with addition as tensor product, and whose morphisms are boolean functions. The monoidal product of this category is in fact the cartesian product, making this category cartesian.

While *parametrised* boolean functions are of course exactly the boolean functions, by distinguishing the p parameters from the a inputs we mean to declare our intent: our goal is to learn a map $\mathbb{Z}_2^a \to \mathbb{Z}_2^b$ which approximates a given dataset of input/output *examples* of the type $(x,y) \in (\mathbb{Z}_2^a, \mathbb{Z}_2^b)$. To do this, we must choose a particular *model*: a function $f : \mathbb{Z}_2^{p+a} \to \mathbb{Z}_2^b$. We then use our machine learning algorithm to search for a set of parameters $\theta \in \mathbb{Z}_2^p$ such that $f(\theta, -) : \mathbb{Z}_2^a \to \mathbb{Z}_2^b$ approximates the dataset well.

Example 2. Suppose we wish to learn a boolean function $\mathbb{Z}_2 \to \mathbb{Z}_2$ with no prior knowledge of the dataset. One choice of model is the function $\texttt{eval} : \mathbb{Z}_2^{2+1} \to \mathbb{Z}_2$, defined as $\texttt{eval}(\theta, x) \mapsto \theta_x$. That is, the function of 2 parameters which maps the data bit x to θ_0 if $x = 0$, and θ_1 if $x = 1$. In this case, our parameters represent a *truth table*: i.e., they extensionally specify the $\mathbb{Z}_2^a \to \mathbb{Z}_2^b$ function we are learning.

Remark 3. An interesting property of the eval model is that, because there are a finite number of boolean functions $\mathbb{Z}_2^a \to \mathbb{Z}_2^b$, the entire function space can be represented with $2^a b$ parameters. Clearly, this makes it suitable for only small a, but [20] demonstrate it can be profitably used as a compositional building block for larger models.

In order to apply Reverse Derivative Ascent to boolean functions, we will need to use the reverse differential operator of a related category, which we now introduce.

[1] Note that our implementation represents circuits in terms of the corresponding boolean functions: this is just a presentation choice, because the category of boolean functions and boolean circuits are isomorphic. We refer to Section 4.2.1 for a more comprehensive discussion on the implementation.

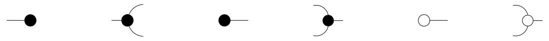
Definition 4. Following [3], let $\mathbf{Poly}_{\mathbb{Z}_2}$ be the category with objects the natural numbers, and with morphisms $p : a \to b$ being b-tuples of polynomials in a variables. That is,

$$p = \langle p_1(\vec{x}), p_2(\vec{x}), \ldots, p_b(\vec{x}) \rangle$$

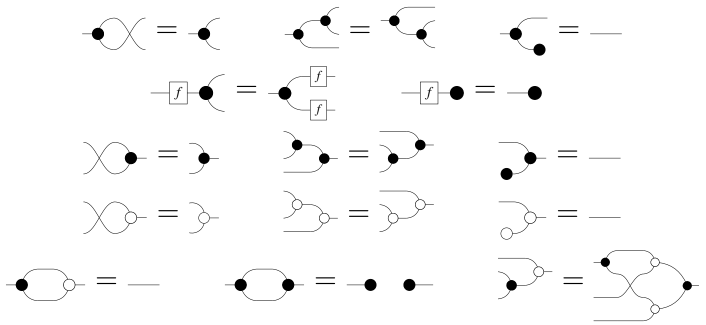
with components $p_i(\vec{x}) \in \mathbb{Z}_2[x_1, \ldots, x_a]$, the polynomial ring in a variables over $\mathbb{Z}_2$. Composition of morphisms is the composition of polynomials as in [3], where the composition $a \xrightarrow{p} b \xrightarrow{q} c$ is the polynomial given by $q(p_1(\vec{x}), p_2(\vec{x}), ..., p_b(\vec{x}))$

In order to make the relationship between **BoolFun** and $\mathbf{Poly}_{\mathbb{Z}_2}$ clear, we will now recall graphical presentations of both. Boolean functions have a well-known graphical representation as *boolean circuits*. This correspondence can be made formal by establishing an isomorphism between **BoolFun** and a category **BoolCirc** whose morphisms are (open) boolean circuits, see [12, section 4]. Furthermore, the morphisms of **BoolCirc** can be pictured as the *string diagrams* [17] freely generated by a certain signature and equations. Similarly, morphisms of $\mathbf{Poly}_{\mathbb{Z}_2}$ have a graphical representation as *polynomial circuits*, which is obtained by relaxing one of the equations of **BoolCirc**. As we will exploit such graphical representations in our developments, we recall them below.

Definition 5. We denote with **BoolCirc** the symmetric strict monoidal category whose objects are the natural numbers and whose morphisms are the string diagrams freely generated by generators

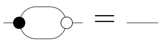
 (1)

and, for all morphisms f, equations



(2)

We call *circuits* the string diagrams freely obtained by the generators in (1) and quotiented by the laws of symmetric monoidal categories. When we say *boolean* circuits, however, we mean morphisms of **BoolCirc**; i.e., those circuits quotiented by equations (2).

Definition 6. We denote by **PolyCirc** the symmetric strict monoidal category whose objects are the natural numbers, and whose morphisms are the string diagrams freely generated by generators (1) and equations (2) minus . We denote this set of axioms as $\mathscr{A}$, and call the morphisms of this category *polynomial circuits*.

Note the equational theories of both Boolean and polynomial circuits yield that (,) and (,) form two commutative monoids and (,) a commutative comonoid. In fact, the comonoid structure makes both **BoolCirc** and **PolyCirc** into cartesian categories.

For boolean circuits, one may think operationally of as the XOR gate, as the AND gate, and as copy. This intuition is at the basis of the interpretation functor $[\![\cdot]\!]^{\mathbb{B}} \colon \mathbf{BoolCirc} \to \mathbf{BoolFun}$ of boolean circuits as boolean functions. Saying that (1) and (2) present **BoolFun** amounts to the following statement.

Proposition 7 ([12]). $[\![\cdot]\!]^{\mathbb{B}} :$ **BoolCirc** → **BoolFun** is an isomorphism of symmetric monoidal categories.

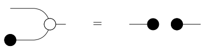
Corollary 8. $[\![c]\!]^{\mathbb{B}} = [\![d]\!]^{\mathbb{B}}$ if and only if $c \overset{(2)}{=} d$.

Similarly for *polynomial* circuits, we may think of and respectively as the two-variable polynomials $x_1 + x_2$ and $x_1 x_2$. Saying that generators (1) and equations $\mathscr{A}$ present **PolyCirc** amounts to the following statement about the interpretation functor $[\![\cdot]\!]^{\mathbb{P}}$:

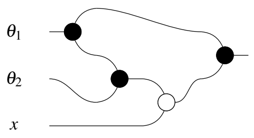
Proposition 9. $[\![\cdot]\!]^{\mathbb{P}} :$ **PolyCirc** → $\mathbf{Poly}_{\mathbb{Z}_2}$ is an isomorphism of symmetric monoidal categories.

Corollary 10. $[\![f]\!]^{\mathbb{P}} = [\![g]\!]^{\mathbb{P}}$ if and only if $f \overset{\mathscr{A}}{=} g$.

Proof. The key idea is that hom-sets $\mathbf{Poly}_{\mathbb{Z}_2}(a,b)$ and $\mathbf{PolyCirc}(a,b)$ have the structure of the free module over the polynomial ring $\mathbb{Z}_2[x_1 \dots x_a]$, and so there exists an isomorphism between them. See Appendix A for the full proof. □

Remark 11. We note that the axiom  from [12, Figure 40] is redundant, and can be derived from the others. This is because for all $n \in \mathbb{N}$, the hom-sets $\mathbf{BoolFun}(n,1)$ have ring structure where $0x = 0$ is an elementary property. We discuss this in more detail in Appendix A.

Example 12. Continuing our example of the eval : $\mathbb{Z}_2^{2+1} \to \mathbb{Z}_2$ function from Example 2, we show its corresponding boolean circuit below, with its inputs labeled. Note that this circuit can be equally interpreted as a morphism of $\mathbf{Poly}_{\mathbb{Z}_2}$, namely as the single 3-variable polynomial $\langle \theta_1 + (\theta_1 + \theta_2)x \rangle$.



(3)

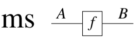
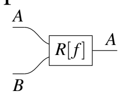
3 Applying the Reverse Derivative to Boolean Circuits

In order to define our machine learning algorithm, we need a notion of *reverse derivative* for boolean circuits. To this aim, we follow a principled approach by recalling reverse differential categories [3], which axiomatise the notion of a reverse differential combinator for categorical morphisms. More concretely, we will translate the reverse derivative combinator of $\mathbf{Poly}_{\mathbb{Z}_2}$ given in [3] to the graphical setting of **PolyCirc**, and show how we can exploit the syntactic similarity with *boolean* circuits in order to apply it to morphisms of **BoolCirc**. However, we will see that this does not make **BoolCirc** a reverse derivative category, and applying the reverse derivative in this way requires a safety condition which we will introduce.

Definition 13. (from [3]) A *reverse differential category* is a category which is

(i) cartesian

(ii) left-additive, meaning that each object a is canonically equipped with a commutative monoid structure $(+_a : a \times a \to a, 0_a : I \to a)$.

(iii) equipped with a *reverse differential combinator* which maps morphisms  to reverse derivatives  . obeying the axioms RD.1 through RD.7 of [3, Section 3].

Intuitively, $R[f]$ approximately computes the change in input necessary to achieve a given change of output for a function f. For example, suppose we have a parametrised boolean function $f : \mathbb{Z}_2^{p+a} \to \mathbb{Z}_2^b$ whose predictions we denote $\hat{y} = f(\theta, x)$. We may have some observed data $(x, y) \in (\mathbb{Z}_2^a, \mathbb{Z}_2^b)$ that disagree with our predictions, i.e., where $\hat{y} \neq y$, and we wish to adjust the parameters of our model θ to better match our observations. The reverse derivative allows us to compute a *change* [2] in parameters δ_θ so that $f(\theta + \delta_\theta, x)$ is a better prediction than $f(\theta, x)$ This intuition is exactly the basis for *reverse derivative ascent*, which we describe in section 4.

Our next goal is to show that we can apply a notion of reverse derivative to morphisms of **BoolCirc**. We establish some preliminary intuition through an example.

Example 14. By directly translating the definition of reverse derivative combinator for morphisms of $\mathbf{Poly}_{\mathbb{Z}_2}$ to boolean functions $f : \mathbb{Z}_2^a \to \mathbb{Z}_2^b$, we obtain the a-tuple of $(a+b)$-variable functions [3]

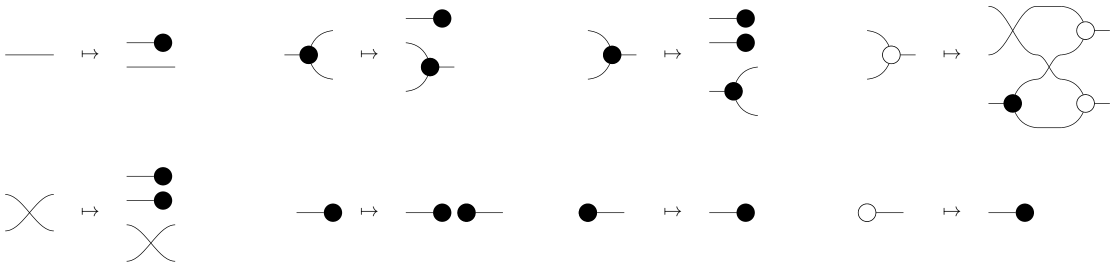
$$\left\langle \sum D_1[f](x) \cdot \delta_y, \ldots, \sum D_a[f](x) \cdot \delta_y \right\rangle$$

where we use $\cdot$ to denote pointwise multiplication of bitvectors, $\sum x$ to denote the sum of vector components, and $D_i[f]$ is the ith partial derivative of f, as defined in [15]: $D_i[f](x) = f(x) + f(x + e_i)$, with e_i the ith basis vector, whose entries are 0 except for the ith, which is 1.

The above indeed allows us to learn the parameters of boolean circuits from data [4] but has one major flaw: efficiency. Computing it requires $i+1$ evaluations of f, and in models with just a moderate number of parameters and/or where f is expensive to compute, this quickly becomes intractable. However, this definition is still useful to take the reverse derivative of a 'black box' function whose symbolic form is not known, such as a function from a software library.

We will now develop a more efficient approach for boolean functions: the key step is the introduction of $\widetilde{R}$, a syntactic operator defined inductively on circuits. However, we will see that this operation does not respect the equations of **BoolCirc**, and so we introduce a safety condition restricting us to those circuits on which $\widetilde{R}$ is well defined. Finally, we show that because every circuit has a safe equivalent in **BoolCirc**, we are able to define an operator for **BoolCirc** which coincides with the reverse derivative of $\mathbf{Poly}_{\mathbb{Z}_2}$.

Definition 15. For each circuit $f : a \to b$, we define the operator $\widetilde{R}[f] : a + b \to a$ inductively on generators (1), composition, and monoidal product. Since each generator represents a specific morphism of $\mathbf{Poly}_{\mathbb{Z}_2}$, we must define $\widetilde{R}$ on generators as



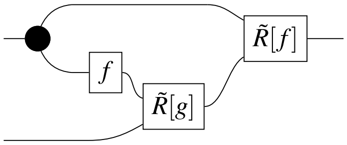
(4)

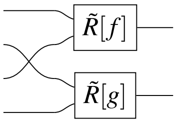
[2] Note that reverse derivatives compute the changes in all inputs, so for a parametrised boolean circuit, this includes both parameter and data inputs.

[3] We note that this definition is essentially the same as the definition of the reverse differential combinator given by [3] for polynomials over a semiring except for the definition of D_i, for which we use the partial derivatives of boolean functions given by [15].

[4] Indeed, we expose it in our Haskell library as the function `RDA.ReverseDerivative.rdiffB`

Following axioms RD.5 and RD.4 of [3, Definition 13], we take $\widetilde{R}$ on composition and monoidal product of circuits as follows:

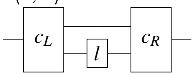
$\tilde{R}[fg] \mapsto$ 

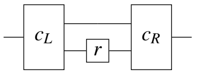
$\tilde{R}[f \otimes g] \mapsto$ 

. (5)

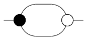
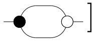
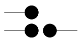
Strictly speaking, since **PolyCirc** is a cartesian category, the definition of $\widetilde{R}[f \otimes g]$ is only implied by RD.4, and so we verify that this definition indeed respects the axioms $\mathscr{A}$ of **PolyCirc**.

Lemma 16. $\widetilde{R}$ is well-defined for circuits modulo $\mathscr{A}$, that is, $c \overset{\mathscr{A}}{=} d$ implies $\widetilde{R}[c] \overset{\mathscr{A}}{=} \widetilde{R}[d]$.

Proof. It suffices to check the statement on c and d that are equal modulo a single axiom $\langle l, r\rangle$ in $\mathscr{A}$. This means that, modulo the laws of symmetric monoidal categories, c can be factorised as 

and d as 

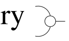
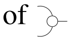
. Thus $\tilde{R}[c] = \tilde{R}[c_L\,;(id \otimes l)\,; c_R]$ and $\tilde{R}[d] = \tilde{R}[c_L\,;(id \otimes r)\,; c_R]$. Unravelling these circuits according to Definition 15, one may observe that in order to prove that $\tilde{R}[c] = \tilde{R}[d]$ we only need to check that $\tilde{R}[l] = \tilde{R}[r]$. This can be verified exhaustively for all $\langle l, r\rangle \in \mathscr{A}$. □

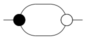
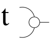
Consequently, it is clear that this syntactic definition of $\widetilde{R}$ is equivalent to the reverse derivative R of **PolyCirc** and, by isomorphism, $\mathbf{Poly}_{\mathbb{Z}_2}$. However, this definition is *not* compatible with boolean circuits: although we have the axiom  $=$ —, we can derive $\widetilde{R}[$$] =$  and $\widetilde{R}[-] =$, which are clearly not equal. Indeed, Lemma 16 highlights that this axiom is the *only* problematic one.

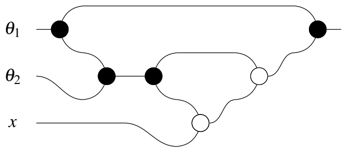
To address this issue, we now introduce a condition called safety, and show that $\widetilde{R}$ always respects (2) when applied to safe circuits. In essence, the following series of results give us a recipe to take the reverse derivative of a boolean circuit, even though **BoolCirc** does not form a reverse derivative category.

3.1 Safety

Safety can be succinctly defined by regarding a boolean circuit combinatorially as a directed graph.[5]

Definition 17. We say a circuit c is *safe* if, for every  generator in c, the two input ports of  are not reachable from the same input port of c.

Example 18. The circuit  is not safe, whereas (3) is safe. The following circuit, which is equivalent in **BoolCirc** to (3), is not safe. Indeed, both inputs of the rightmost  generator are reachable from θ_1 (and actually also from θ_2).

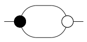


(6)

As witnessed by (3) and (6), it is actually possible to show that

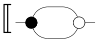
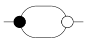
Lemma 19. For each boolean circuit c, there is a safe boolean circuit d such that $c \overset{(2)}{=} d$.

[5] This can be made completely formal by interpreting string diagrams as (directed) hypergraphs with boundaries [2, 21], where generators form hyperedges and wires connecting them are the nodes. In this context, reachability between wires (as in Definition 17 below) can be defined as the existence of a forward path between the corresponding nodes in the hypergraphs.

Proof. The idea is that one may put circuits in a canonical form, so that then it is straightforward to eliminate all the unsafe paths by iteratively applying  $=$ — as a rewrite rule. See Appendix A for the full proof. □

By virtue of Lemma 19, in order to show that $\widetilde{R}$ yields a well-defined operator on the whole of **BoolCirc**, it suffices to show that it is well-defined on safe circuits. To do so, the following is the key intermediate lemma.

Lemma 20. If two circuits c and d are safe and $c \overset{(2)}{=} d$, then $c \overset{\mathscr{A}}{=} d$.

Proof. We give an overview of the structure of the argument and refer to Appendix A for the full details. The proof relies on the polynomial interpretation of circuits, $[\![\cdot]\!]^{\mathbb{P}}$. Under this interpretation, $[\![$  $]\!]^{\mathbb{P}} = \langle x^2\rangle$ and $[\![$ — $]\!]^{\mathbb{P}} = \langle x\rangle$. Thus  $=$ — is not sound under this interpretation, because $\langle x^2\rangle \neq \langle x\rangle$. On the other hand, we know that $c \overset{\mathscr{A}}{=} d$ if and only if $[\![c]\!]^{\mathbb{P}} = [\![d]\!]^{\mathbb{P}}$ by Corollary 10. By definition, this is the same as saying that $c \overset{(2)}{=} d$ if and only if $[\![c]\!]^{\mathbb{P}} = [\![d]\!]^{\mathbb{P}}$ modulo the equation $x^2 = x$.

Now, one can prove that, for any safe circuit c, $[\![c]\!]^{\mathbb{P}}$ does not contain any squared term (Lemma 34). Thus, coming to c and d as in the statement of the lemma, because c and d are such that $c \overset{(2)}{=} d$, it follows that $[\![c]\!]^{\mathbb{P}} = [\![d]\!]^{\mathbb{P}}$ modulo the equation $x^2 = x$. Because c and d are safe, they do not contain any squared terms, and thus $[\![c]\!]^{\mathbb{P}} = [\![d]\!]^{\mathbb{P}}$. By completeness of $\mathscr{A}$ with respect to $[\![\cdot]\!]^{\mathbb{P}}$, we conclude that $c \overset{\mathscr{A}}{=} d$. □

We can now conclude that

Proposition 21. $\widetilde{R}$ is well-defined on safe circuits modulo (2), that is, for c and d safe, if $c \overset{(2)}{=} d$ then $\widetilde{R}[c] \overset{(2)}{=} \widetilde{R}[d]$.

Proof. Suppose we have two safe circuits c, d such that $c \overset{(2)}{=} d$. By Lemma 20 we know that $c \overset{\mathscr{A}}{=} d$, and therefore we have $\widetilde{R}[c] \overset{\mathscr{A}}{=} \widetilde{R}[d]$ by Lemma 16. Finally, because $\mathscr{A} \subseteq (2)$, we have that $\widetilde{R}[c] \overset{(2)}{=} \widetilde{R}[d]$. □

We can now define an operator R of boolean circuits which computes the reverse derivative.

Definition 22. Let f be a morphism of **BoolCirc**, i.e. a (2)-equivalence class of boolean circuits. Let c be a safe circuit in such class, which exists by Lemma 19. Define $R[f]$ as the (2)-equivalence class of $\widetilde{R}[c]$. Since c is safe, we know that R is well defined thanks to Proposition 21.

Note that this definition is a minor abuse of notation, because R does *not* make **BoolCirc** a reverse derivative category. This is because the safety condition is not compositional, and thus cannot satisfy axiom RD.5. Nevertheless, we are still able to use R to learn the parameters of boolean functions, as we demonstrate in the following sections.

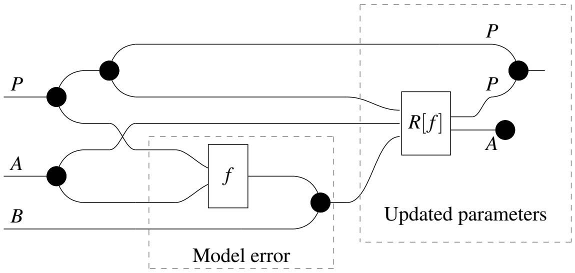
4 Reverse Derivative Ascent

4.1 Reverse Derivative Ascent Algorithm

We now introduce our machine learning algorithm, *reverse derivative ascent*. The definition refers to the category **BoolCirc**, as boolean circuits are our motivating example. However, our formulation makes sense in any reverse differential category.

We proceed in two parts: the inner 'step' of the algorithm, which we call `rdaStep`, and the outer 'iteration' of `rdaStep`, which is `rda`.

Definition 23. Let $f : p+a \to b$ be a boolean circuit in **BoolCirc**, thus computing a parametrised boolean function with p parameters. We define $\texttt{rdaStep}_f$ as

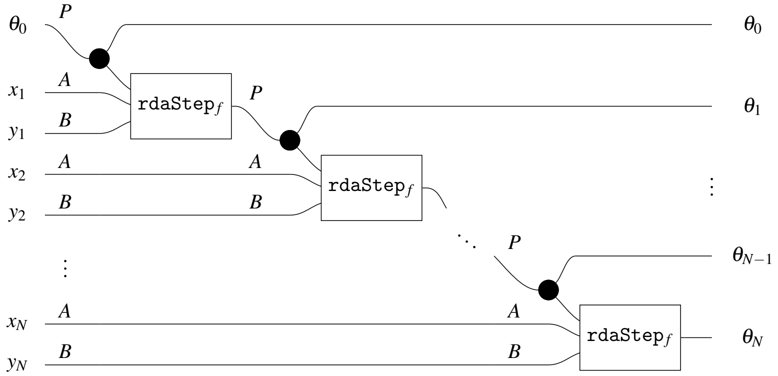


(7)

$\texttt{rdaStep}_f$ represents a single iteration of `rda`. Its function is to compute a new parameter vector θ' for a single labelled dataset example $(x,y) \in (\mathbb{Z}_2^a, \mathbb{Z}_2^b)$. We highlight two important parts of `rdaStep`. First, it computes the *model error* $\delta_y := f(\theta,x)+y$: the difference in model prediction to true label. Secondly, it uses $R[f]$ to compute a *change in parameters* $\delta_\theta := R[f](\theta,x,\delta_y)\pi_0$ [6] such that $f(\theta+\delta_\theta,-)$ will more closely approximate the example datum.

Of course, we would like to update our parameters multiple times: this is the *ascent* part of reverse derivative ascent. In Haskell `rda` is simply the `scanl` operation over `rdaStep`, but we define `rda` as a circuit to emphasize its generality as a morphism of a reverse derivative category.

Definition 24. Let $n \in \mathbb{N}$, and let $(x_i, y_i) \in (\mathbb{Z}_2^a, \mathbb{Z}_2^b)$, denote a sequence of examples with $0 < i \leq n$. $\texttt{rda}_f$ is defined as



(8)

Remark 25. In general, there is no need for elements (x_i, y_i) to be in direct correspondence with elements of the dataset. Commonly, the sequence of examples ‘shown’ to the algorithm will be shuffled with repetitions [16, 6.1].

4.2 Empirical Results

We now show empirical results of our method (Table 1), which suggest that our algorithm is genuinely able to learn useful functions from real-world data. The full source code for running these experiments is available at `http://catgrad.com/p/reverse-derivative-ascent`. We begin with a brief discussion of our implementation.

[6] Following [3], we write composition left-to-right to mimic diagrammatic order.

Table 1: Empirical Results

Dataset	Model	Label Encoding	Accuracy %
Iris (2-class)	eval : $16+4 \to 1$	binary	98.0%
Iris (2-class)	eval : $16+4 \to 2$	one-hot	98.0%
Iris	eval : $16+4 \to 2$	binary	73.3%
Iris	eval : $16+4 \to 3$	one-hot	73.3%
MNIST (2 class)	pseudoLinear : $784+784 \to 1$	binary	99.2%

4.2.1 Implementation Details

The purpose of our implementation is to specify and evaluate circuits as machine learning models. A boolean circuit $a \to b$ is represented by a term of the datatype `a :-> b`; more complex circuits are built by composition and tensoring from the primitives of (1). Note that, in our implementation, such primitives already come interpreted as the corresponding boolean functions, exploiting the isomorphism between **BoolCirc** and **BoolFun** (Proposition 7). This is just a presentation choice, which spares us the need to define a separate syntax and interpreter. In the future, we plan to enhance the flexibility of our tool by making these two components distinct.

In fact, our datatype `:->` is a *pair* of a circuit and its reverse derivative: constructing a circuit simultaneously constructs its reverse derivative precisely as in (5).[7] In this way, the reverse derivative is built up compositionally from smaller parts, and therefore to compute the reverse derivative we need only to extract the second element of the pair, for which we provide the `rdiff` function.

As we have seen in Section 3, composing reverse derivatives in this way is only valid for *safe* circuits. This prototype version of the code does not implement the procedure described in Appendix A to extract a safe circuit, and so we provide a second method to compute reverse derivatives: the brute-force `rdiffB` function (as described in Example 14). Note that this method can be applied even to unsafe circuits, but is significantly less efficient compared to the compositional `rdiff` as defined in Definition 15. For example, in our experiment code we consider the `eval` model–an instance of which we show in Example 12. For a-dimensional input, the model has 2^a parameters, and so computing `rdiffB eval` requires running `eval` an exponential number of times. By comparison, $R[$`eval`$]$ (as computed by `rdiff eval`) is a circuit whose size is within a constant factor of `eval`, and whose result needs to be computed just once.

To showcase the difference, in the two experiments which follow, we use `rdiff` for compositionally for the Iris model–since it is safe–but use `rdiffB` for the MNIST model. In the latter case, the number of parameters is equal to the number of inputs, so this method is not too computationally demanding.

We discuss further avenues for improvement to our prototype implementation in Section 5.

4.2.2 Iris Dataset & Model

The Iris dataset [5] is a simple example of a classification problem, and is frequently used for pedagogical purposes, e.g. in [6]. It consists of 150 labelled examples of three types of iris flower. Each example consists of four measurements of the flower petal and sepal sizes, so we have the dataset of examples $(\tilde{x},\tilde{y}) \in (\mathbb{R}^4, \{\text{Setosa}, \text{Versicolor}, \text{Virginica}\})$.

[7] Interestingly, this pairing is the way in which the reverse derivative construction can be made functorial. See [3, Proposition 31] for details.

We run two experiments with this dataset, using our running example of the `eval` model. We first tackle the simpler problem of the sub-dataset consisting of the labeled examples for classes Setosa and Versicolor, which we call 'Iris (2-class)' in Table 1, and then show results for the full 3-class problem. We also run two variations of each of these two experiments, corresponding to different ways of encoding the labels: the 3-bit one-hot encoding[8], and the encoding of labels as binary numbers. In all experiments, we preprocess this data by *normalizing* and *rounding* each feature $\tilde{x}_i \in \mathbb{R}$ into a single bit $x_i \in \mathbb{Z}_2$.[9] For the n-class problem, this gives us a dataset of examples $(x,y) \in (\mathbb{Z}_2^4, \mathbb{Z}_2^n)$ for the one-hot encoding, and $(x,y) \in (\mathbb{Z}_2^4, \mathbb{Z}_2^{\lceil \log_2(n) \rceil})$ for the binary encoding.

4.2.3 MNIST Dataset & Model

MNIST [13] is an image classification dataset widely used as a benchmark in machine learning (see e.g., [4]). It consists of 60000 examples of images of handwritten numeric digits (0 to 9), with each image consisting of 28×28 greyscale pixels encoded as bytes. The dataset therefore consists of examples $(\tilde{x}, \tilde{y}) \in (\{0..255\}^{28\times 28}, \{0..9\})$.

We do not tackle the full 10-class problem, but leave it for future work. Instead, we restrict ourselves to the subset of classes $\{0,1\}$. While this means we cannot compare our method to the state of the art on this benchmark, we believe it demonstrates that our method is indeed capable of learning. As in the Iris data, we also *binarize* the pixels of the dataset by normalisation and rounding, to give our 'binarized' dataset of examples $(x,y) \in (\mathbb{Z}_2^{28\times 28}, \mathbb{Z}_2)$

Clearly the dimensionality of this problem is too large to use `eval`, so we instead use a model `pseudoLinear` : $(28 \times 28) + (28 \times 28) \to 1$, so named because its structure is loosely inspired by the linear layers of neural networks. We give only a brief informal description of this model here (for technical details, see the experiment code we release with this paper [10])

Essentially, the model learns a 'feature mask', which is simply a bitmap image that is pointwise multiplied with the input. If the resulting bitvector has fewer than 25% as many 1 bits as the mask, the model returns 1. The intuition is that the model should learn the 'average' handwritten 0 digit, and compare it with inputs. In the two-class case this is a fair assumption, since images of 1 and 0 are typically very different, but it is unlikely to generalise well.

4.2.4 Discussion of Results

From Table 1, we can see that the `eval` model is able to learn a near-perfect classifier for the 2-class problem, but fares poorly on the full problem. This is because our preprocessing essentially limits the model to fixed, axis-aligned decision boundaries. Since the Setosa and Versicolor classes are clearly separable when plotted, this works well, but the Versicolor and Virginica classes are not. We also note that because `eval` is essentially a lookup table, the label encoding has no effect on model accuracy. This suggests that `eval` may be useful as an 'output unit' in larger models.

Finally, we note that our MNIST model, while only classifying a subset of the full problem, returns fairly good results. To make an apples-to-oranges comparison, the approach of [4] gives a similar accuracy of 99.13%. However, such a comparison is to be taken with a grain of salt: the full MNIST problem

[8] `one-hot` refers to the standard practice of encoding the ith class label of n total as a vector with zero entries except for the ith. See e.g., [1, section 3.3]

[9] This is essentially throwing away as much of the information of the dataset as possible: we map each feature to a simple 'high' or 'low' value

[10] `https://github.com/statusfailed/act-2020-experiments`

is of course much more difficult than the version we tackle here.

5 Discussion and Future Work

In this paper, we saw how the categorical axiomatisation of reverse derivative can be used to define a general ‘gradient’ based algorithm for machine learning. Further, we showed how our algorithm can be used to learn parameters of a novel model class: boolean circuits. However, there are many opportunities for future work, which we broadly classify into two parts.

Empirical Work The first task is to discover principles for building effective parametrised circuit models. While a number of compositional building blocks for neural network models have been discovered and studied, the same is not true for parametrised boolean circuits. One exciting challenge is to understand whether neural network architectures can be translated to the setting of circuits

Furthermore, although our empirical results show our algorithm is certainly able to learn parameters from data, a new machine learning method would typically be expected to show results on the *full* MNIST problem, as well as other image processing benchmarks like CIFAR [11]. Therefore, some empirical study of circuit architectures with respect to these benchmarks will have to be undertaken.

As mentioned in Section 4.2.1, another important point is to enhance our implementation. First, we intend to clearly separate between boolean circuits and their interpretations as boolean functions, so that other semantic interpretations are possible. Second, we plan to implement our procedure to turn a circuit into its safe equivalent, and study its complexity.

Theoretical Work One avenue for theoretical work is to demonstrate the use of Reverse Derivative Ascent on categories other than boolean circuits. For example, when interpreted in the category of natural numbers and morphisms $a \to b$ the smooth maps $\mathbb{R}^a \to \mathbb{R}^b$, our method is similar to stochastic gradient descent (SGD) [16], with the following differences. Firstly, computing the model error ((7)) means computing the *difference* between true label and model prediction, but in $\mathbb{Z}_2$ this coincides with addition because elements are self-inverse. Secondly, SGD has a notion of *learning rate*: a constant multiplied by the parameter change which prevents the algorithm ‘overshooting’ the optimal parameter value. Thirdly, we have no explicit *loss function*, which is important to discover the conditions under which guarantees of convergence exist. By comparison, several different guarantees of convergence are known for different variants of gradient descent as used in neural networks (see e.g., [16, p. 2].), although in some cases tweaks such as slowly decreasing the learning rate are required to make such guarantees, and prevent oscillation around local minima.

Another setting of interest is boolean circuits with notions of *feedback*–something which has already received attention in the literature [18] [19]. Characterising these differences between settings may help to understand gradient methods in a more general light.

It will also be important to relate our work to existing category theoretic views of gradient-based methods such as [8] [7]. In particular, we believe our method is a special case of [7]. Concretely, we note that for two parametrised boolean functions $f : p+a \to b, g : q+b \to c$, taking the reverse derivative of their ‘parametrised composition’ $(\mathrm{id} \times f)g : q+p+a \to c$ is exactly the composite update-request morphism from their formalism.

Acknowledgements We are grateful to the reviewers for their insightful comments and remarks. We would also like to thank David Sprunger and Liviu Pirvan for several helpful discussions.

References

[1] Christopher M. Bishop (2006): *Pattern recognition and machine learning*. Information science and statistics, Springer, New York, doi:10.978.038731/0732.

[2] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski & Fabio Zanasi (2016): *Rewriting modulo symmetric monoidal structure*. Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '16, pp. 710–719, doi:10.1145/2933575.2935316.

[3] Robin Cockett, Geoffrey Cruttwell, Jonathan Gallagher, Jean-Simon Pacaud Lemay, Benjamin MacAdam, Gordon Plotkin & Dorette Pronk (2019): *Reverse derivative categories*. arXiv:1910.07065 [cs, math].

[4] Matthieu Courbariaux, Yoshua Bengio & Jean-Pierre David: *BinaryConnect: Training Deep Neural Networks with binary weights during propagations*, p. 9.

[5] Dheeru Dua & Casey Graff (2017): *UCI Machine Learning Repository*.

[6] Richard O. Duda, Peter E. Hart & David G. Stork (2000): *Pattern Classification (2nd Edition)*. Wiley-Interscience, USA.

[7] Brendan Fong, David I. Spivak & Rémy Tuyéras (2019): *Backprop as Functor: A compositional perspective on supervised learning*. arXiv:1711.10455 [cs, math].

[8] Bruno Gavranovic: *Learning Functors using Gradient Descent*, p. 12.

[9] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv & Yoshua Bengio: *Binarized Neural Networks*, p. 9.

[10] Nathan Jacobson (2012): *Basic Algebra I: Second Edition*. Courier Corporation.

[11] Alex Krizhevsky (2009): *Learning Multiple Layers of Features from Tiny Images*, p. 60.

[12] Yves Lafont (2003): *Towards an algebraic theory of Boolean circuits*. Journal of Pure and Applied Algebra 184(2-3), pp. 257–310, doi:10.1016/S0022-4049(03)00069-0.

[13] Yann Lecun, Léon Bottou, Yoshua Bengio & Patrick Haffner (1998): *Gradient-Based Learning Applied to Document Recognition*. In: Proceedings of the IEEE, pp. 2278–2324.

[14] Rajat Raina, Anand Madhavan & Andrew Y. Ng (2009): *Large-scale deep unsupervised learning using graphics processors*. In: Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09, ACM Press, Montreal, Quebec, Canada, pp. 1–8, doi:10.1145/1553374.1553486.

[15] A. Martín del Rey, G. Rodríguez Sánchez & A. de la Villa Cuenca (2012): *On the boolean partial derivatives and their composition*. Applied Mathematics Letters 25(4), pp. 739–744, doi:10.1016/j.aml.2011.10.013.

[16] Sebastian Ruder (2017): *An overview of gradient descent optimization algorithms*. arXiv:1609.04747 [cs].

[17] Peter Selinger (2010): *A survey of graphical languages for monoidal categories*. arXiv:0908.3347 [math] 813, pp. 289–355, doi:10.1007/978-3-642-12821-9-4.

[18] David Sprunger & Bart Jacobs (2019): *The differential calculus of causal functions*, p. 15.

[19] David Sprunger & Shin-ya Katsumata (2019): *Differentiable Causal Computations via Delayed Trace*. In: 2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), IEEE, Vancouver, BC, Canada, pp. 1–12, doi:10.1109/LICS.2019.8785670.

[20] Erwei Wang, James J. Davis, Peter Y. K. Cheung & George A. Constantinides (2019): *LUTNet: Rethinking Inference in FPGA Soft Logic*. arXiv:1904.00938 [cs, stat].

[21] Fabio Zanasi (2017): *Rewriting in Free Hypergraph Categories*. Electronic Proceedings in Theoretical Computer Science 263, pp. 16–30, doi:10.4204/EPTCS.263.2.

[22] Ivan Zhegalkin (1927): *Sur le calcul des propositions dans la logique symbolique*.

A Polynomial Interpretation of Boolean Circuits

In subsection 3.1 we used the interpretation of circuits with axioms $\mathscr{A}$ as morphisms of $\mathbf{Poly}_{\mathbb{Z}_2}$. We now make this interpretation precise. We will discuss two results for these polynomial circuits: soundness and completness of their interpretation $[\![\cdot]\!]^{\mathbb{P}}$, and a canonical form.

A.1 Soundness and Completeness

To show the existence of an isomorphism between **PolyCirc** and $\mathbf{Poly}_{\mathbb{Z}_2}$, we will show that both categories' hom-sets have the structure of the free module over the polynomial ring. For this, we must recall the definition of a free module:

Definition 26. Following [10, p. 170], let S be a ring. The free module S^b is the cartesian product of b elements of S, i.e. $S^b = \langle p_1, p_2, ..., p_b \rangle$, with addition defined pointwise, $\langle p_1, p_2, ...p_b \rangle + \langle q_1, q_2, ...q_b \rangle = \langle p_1 + q_1, p_2 + q_2, ...p_b + q_b \rangle$ a zero element $\mathbf{0} = \langle 0, 0, ..., 0 \rangle$ and scalar multiplication $s \langle p_1, p_2, ..., p_b \rangle = \langle sp_1, sp_2, ...sp_b \rangle$

It is clear that the hom-sets of $\mathbf{Poly}_{\mathbb{Z}_2}$ have this structure

Proposition 27. Hom-sets $\mathbf{Poly}_{\mathbb{Z}_2}(a,b)$ have the structure of the free module S^b with S the polynomial ring $S = \mathbb{Z}_2[x_1, ..., x_a]$.

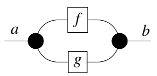
Proof. Immediate from the definition of $\mathbf{Poly}_{\mathbb{Z}_2}$ □

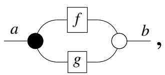
Furthermore, hom-sets of **PolyCirc** also have this structure. This implies the existence of a module isomorphism between the hom-sets of **PolyCirc** and $\mathbf{Poly}_{\mathbb{Z}_2}$ which is the basis for the functor $[\![\cdot]\!]^{\mathbb{P}}$. We begin, however, with some special case examples.

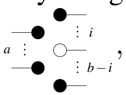
Example 28. The hom-set $\mathbf{PolyCirc}(0,1)$ has the structure of the ring $\mathbb{Z}_2$, with every circuit c equal to ●— or ○—.

Example 29. Each hom-set $\mathbf{PolyCirc}(a,1)$ has the structure of the polynomial ring $\mathbb{Z}_2[x_1, ..., x_a]$, with indeterminates $x_1 \dots x_a$ given by the projections $\pi_1 \dots \pi_a$

Proposition 30. Hom-sets $\mathbf{PolyCirc}(a,b)$ have the structure of the free module $\mathbb{Z}_2[x_1, \dots, x_a]^b$.

Proof. For morphisms $f, g : a \to b$, put addition $f + g =$ 

and multiplication $f * g =$ 

, with the zero element defined as $\mathbf{0} =$ a —● ●— b. one can verify graphically using equations $\mathscr{A}$ that the module axioms hold. [11] If we define the family of b morphisms $e_i :=$ 

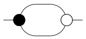
$, 0 < i \leq b$, we can see that it forms a base: each of the generators of Equation 1 can be constructed through addition and scalar multiplication of morphisms e_i and $\mathbf{0}$. □

We are now ready to give the proof of Proposition 9.

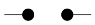
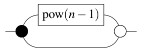
Proof of Proposition 9. By Proposition 27 and Proposition 30, there is a module isomorphism between $\mathbf{Poly}_{\mathbb{Z}_2}(a,b)$ and $\mathbf{PolyCirc}(a,b)$. Further, because the identity-on-objects functor $[\![\cdot]\!]^{\mathbb{P}}$ is defined in terms of this bijection, it is a full and faithful functor, and so $\mathbf{Poly}_{\mathbb{Z}_2} \cong \mathbf{PolyCirc}$. □

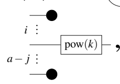
[11] We take *scalar* multiplication of $f : a \to b$ by $g : a \to 1$ as the morphism $f * (g\Delta^*)$, where Δ^* is the unique $1 \to b$ morphism formed by tensor and composition of the diagonal map and identity.

A.2 Canonical Form

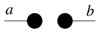
We now give a canonical form for morphisms of **PolyCirc**. This canonical form essentially isolates all occurrences of the axiom  $=$ —, which we can then use to show that all boolean circuits have a safe equivalent in the proof of Lemma 19.

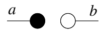
Definition 31. We say a circuit $f : a \to b$ is in canonical form if it can be written as $\langle \mathbf{C}(p_1), \mathbf{C}(p_2), \dots, \mathbf{C}(p_b)\rangle$, where $\langle p_1, \dots, p_b\rangle = [\![f]\!]^{\mathbb{P}}$, and $\mathbf{C}(-)$ is defined on polynomials as follows.

Denote by $\mathrm{pow}(n)$ the morphism defined inductively as $\mathrm{pow}(0) =$  and $\mathrm{pow}(n) =$ 

, and let x_i^k be an arbitrary indeterminate raised to a power $k \in \mathbb{N}$. We define $\mathbf{C}(x_i^k) :=$ 

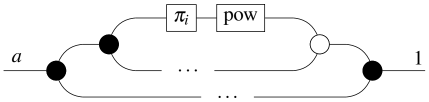
, and note that this is consistent with $[\![\cdot]\!]^{\mathbb{P}}$ in the sense that $\pi_i * \overset{k}{\dots} * \pi_i \overset{\mathscr{A}}{=} \mathbf{C}(x_i^k)$.

We now define $\mathbf{C}(\cdot)$ inductively on polynomials. Either p is a constant, in which case $\mathbf{C}(0) =$ 

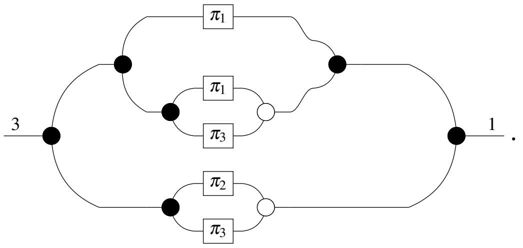
and $\mathbf{C}(1) =$ 

, or $p = m + p'$ and we have $\mathbf{C}(m) + \mathbf{C}(p')$, where m is a monomial and p' a polynomial. A monomial is a product of *distinct* indeterminates raised to powers $x_i^{k_i}, x_j^{k_j}, \dots$, and its canonical form is therefore $\mathbf{C}(x_i^{k_i}) * \mathbf{C}(x_j^{k_j}) * \dots$.

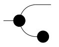
Remark 32. Intuitively, the canonical form can be pictured as



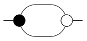
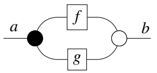
Example 33. Continuing with our running example, we note that the `eval` $: 2+1 \to 1$ circuit can be written as the polynomial $x_1 + x_1x_3 + x_2x_3$ (where parameters are x_1 and x_2), and so its safe canonical form can be written as



.

To see it is equivalent to (3), we can apply the counit law  $=$ —, and then use distributivity.

We are now ready to show the proof of Lemma 19.

Proof of Lemma 19. We show that for each circuit c there is a safe circuit d such that $c \overset{(2)}{=} d$. We begin by noting that with equations (2), we have that $\mathrm{pow}(k) =$ —, which can be seen by repeatedly applying the  $=$ — axiom. Using this, we can see that the canonical form of Definition 31 can be rewritten so that each $\mathrm{pow}(k)$ morphism becomes the identity. Finally, because each product 

in the canonical form is of distinct indeterminates, the rewritten canonical form does not contain any more squared terms, and so is safe. □

To conclude, we show how the condition of safety essentially allows only those circuits with interpretations as Zhegalkin polynomials [22], as used in the proof of Lemma 20.

Lemma 34. If a circuit f is safe, then the polynomial $[\![f]\!]^{\mathbb{P}}$ has only exponents in $\{0,1\}$.

Proof. The combinatorial condition (Definition 17) requires that the inputs of each connect to disjoint sets of inputs. Therefore, the resulting polynomial $[\![f]\!]^{\mathbb{P}}$ only contains multiplications of polynomials $p(\vec{x}_1)p(\vec{x}_2)$ of disjoint sets of variables, so there can be no squared terms in $[\![f]\!]^{\mathbb{P}}$. □