Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 347 (2019) 161–177

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Neural Nets via Forward State Transformation and Backward Loss Transformation

Bart Jacobs[1] David Sprunger[2]

Institute for Computing and Information Sciences (iCIS)
Radboud University Nijmegen
The Netherlands

ERATO Metamathematics for Systems Design Project
National Insitute of Informatics
Tokyo, Japan

Abstract

This article studies (multilayer perceptron) neural networks with an emphasis on the transformations involved — both forward and backward — in order to develop a semantic/logical perspective that is in line with standard program semantics. The common two-pass neural network training algorithms make this viewpoint particularly fitting. In the forward direction, neural networks act as state transformers, using Kleisli composition for the multiset monad — for the linear parts of network layers. In the reverse direction, however, neural networks change losses of outputs to losses of inputs, thereby acting like a (real-valued) predicate transformer. In this way, backpropagation is functorial by construction, as shown in other works recently. We illustrate this perspective by training a simple instance of a neural network.

Keywords: Neural network, backpropagation, multilayer perceptron, state-and-effect triangle, loss transformation

1 Introduction

Though interest in artificial intelligence and machine learning have always been high, the public's exposure to successful applications has markedly increased in recent years. From consumer-oriented applications like recommendation engines, speech face recognition, and text prediction to prominent examples of superhuman performance (DeepMind's AlphaGo, IBM's Watson), the impressive results of machine learning continue to grow.

Though the understandable excitement around the expanding catalog of successful applications lends a kind of mystique, neural networks and the algorithms

[1] Email:bart@cs.ru.nl
[2] Email:sprunger@nii.ac.jp

https://doi.org/10.1016/j.entcs.2019.09.009

which train them are, at their core, a special kind of computer program. One perspective on programs which is relevant in this domain are so-called *state-and-effect triangles*, which emphasize the dual nature of programs as both state and predicate transformers. This framework originated in quantum computing, but has a wide variety of applications including deterministic and probabilistic computations [6].

The common two-pass training scheme in neural networks makes their dual role particularly evident. Operating in the "forward direction" neural networks are like a function: given an input signal they behave like (a mathematical model of) a brain to produce an output signal. This is a form of state transformation. In the "backwards direction", however, the derivative of a loss function with respect to the output of the network is *backpropagated* [8] to the derivative of the loss function with respect to the inputs to the network. This is a kind of predicate transformation, taking a real-valued predicate about the loss at the output and producing a real-valued predicate about the source of loss at the input. The main novel perspective offered by this paper uses such state-and-effect 'triangles' for neural networks. This state-and-effect framework has also been used to understand other artificial representations of human reasoning, including Bayesian networks [7].

In recent years, it has become apparent that the architecture of a neural network is very important for its accuracy and trainability in particular problem domains [3]. This has resulted in a profligation of specialized architectures, each adapted to its application. Our goal here is not to express the wide variety of special neural networks in a single framework, but rather to describe neural networks generally as an instance of this duality between state and predicate transformers. Therefore, we shall work with a simple, suitably generic neural network type called the *multilayer perceptron* (MLP).

We see this paper as one of recent steps towards the application of modern semantic and logical techniques to neural networks, following for instance [1,2]. The main contribution of this paper is that it offers its own representation of network layers as Kleisli maps — as alternative to the representation given in [1]. There, a neural network layer $n \to m$, with a function $\mathbb{R}^n \to \mathbb{R}^m$ as intended meaning, is represented as map $P \times n \to m$, where P is an unspecified, abstract object of parameters. These parameters are typically matrices in $\mathbb{R}^{n \times m}$, representing the linear part of the network layer. The updating of parameters is handled in [1] via an 'update' function $P \times n \times m \to P$, together with a 'request' function $P \times n \times m \to n$ whose role is somewhat mysterious. The sequential (and parallel) composition of such layers can be given in terms of non-trivial string diagrams.

Here we follow a much simpler approach which does not 'internalise' the parameters via parameter objects P. Instead, the linear parts of network layers are represented via Kleisli maps of the multiset monad $\mathcal{M}$. Forward propagation of a state $\boldsymbol{x} \in \mathbb{R}^n$ along a layer is then quite naturally represented via Kleisli composition, involving the familiar linear expressions $\sum_i w_{ij} \cdot x_i$ with weights w_{ij} that are used as scalars for inputs x_i. This paper elaborates this approach, which is 'obvious', from the perspective of monadic computation, and more concrete than the one with abstract parameter objects P. It allows us to describe backpropagation

also concretely, in terms of matrices and derivatives. This part of the story can be compared to predicate transformation in program semantics, which also works in backward direction.

Since the categorical modelling of neural computing is still in its infancy, it is too early to say which one is the 'best' and we do not yet have technical results in that direction. Indeed, it makes sense to leave room for different representations. We expect that these formal approaches to neural networks can be of use as the community develops *explainable* AI, where the goal is to extend automated decisions/classifications with human understandable explanations. In this context, we believe having a diversified pool of techniques will be a valuable resource; we are wary of prematurely attempting to rank by usefulness formal approaches with incomparable aspects.

Outline. In this paper, we begin by describing MLPs, the layers they are composed of, and their forward semantics as a state transformation (Section 2). In Section 3, we give the corresponding backwards transformation on loss functions and use that to formulate backpropagation in Section 4. Finally, in Section 5, we discuss the compositional nature of backpropagation by casting it as a functor, and compare our work in particular to [1].

2 Forward state transformation

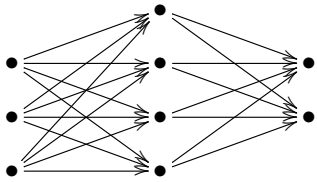
Much like ordinary programs, neural networks are often subdivided into functional units which can then be composed both in sequence and in parallel. These subnetworks are usually called *layers*, and the sequential composition of several layers is by definition a "deep" network[3]. There are a number of common layer types, and a neural network can often be described by naming the layer types and the way these layers are composed.

Feedforward networks are an important class of neural networks where the composition structure of layers forms a directed acyclic graph—the layers can be put in an order so that no layer is used as the input to an earlier layer. A *multilayer perceptron* is a particular kind of feedforward network where all layers have the same general architecture, called a fully-connected layer, and are composed strictly in sequence. As mentioned in the introduction, the MLP is perhaps the prototypical neural network architecture, so we treat this network type as a representative example. In the sequel, we will use the phrase "neural network" to denote this particular network architecture.

More concretely, a layer consists of two lists of nodes with directed edges between

[3] In contrast, the "width" of a layer typically refers to the number of input and output units, which can be thought of as the repeated parallel composition of yet another architecture.

them. For instance, a neural network with two layers may be depicted as follows.



We will represent such a network via special arrows $3 \Rightarrow 4 \Rightarrow 2$, where the numbers 3, 4, and 2 correspond to the number of nodes at each stage. These arrows involve weights, biases, masks, and activations, see Definition 2.1 below. The (forward) semantics of these arrows is given by functions $\mathbb{R}^3 \to \mathbb{R}^4 \to \mathbb{R}^2$. They will be described in greater detail shortly, in Definition 2.3. We first concentrate on individual layers.

In the definition below we shall write $\mathcal{M}(n) = \mathbb{R}^n$ and $\mathcal{P}(n) = \{k \in \mathbb{N} \mid k \subseteq n\}$. In this description of the powerset $\mathcal{P}$ we identify a natural number $n \in \mathbb{N}$ with the n-element subset of numbers $\{0, 1, \ldots, n-1\}$ below n. We shall have more to say about $\mathcal{M}$ and $\mathcal{P}$ in Remark 2.2 below.

Definition 2.1 A single *layer* $n \Rightarrow k$ between natural numbers $n, k \in \mathbb{N}$ is given by three functions:

$$\begin{array}{rl} n+1 \xrightarrow{T} \mathcal{M}(k) & \text{the transition function} \\ n \xrightarrow{M} \mathcal{P}(k) & \text{the mask function} \\ \mathbb{R} \xrightarrow{\alpha} \mathbb{R} & \text{the activation function.} \end{array}$$

The transition function T can be decomposed into a pair $[T_w, T_b]$, where $T_w \colon n \to \mathcal{M}(k)$ captures the weights and $T_b \in \mathcal{M}(k)$ the biases. The mask function $M \colon n \to \mathcal{P}(k)$ captures connections and mutability; it works as follows, for $i \in n$ and $j \in k$.

$$\begin{cases} j \in M(i) & \text{means} \quad \text{there is a mutable connection from node } i \text{ to node } j, \text{ with weight } T(i)(j) \\ j \notin M(i) \text{ and } T(i)(j) = 0 & \text{means} \quad \text{there is } \textit{no} \text{ connection from node } i \text{ to node } j \\ j \notin M(i) \text{ and } T(i)(j) \neq 0 & \text{means} \quad \text{there is a } \textit{non-mutable} \text{ connection from node } i \text{ to node } j, \text{ with weight } T(i)(j). \end{cases}$$

The activation function $\alpha \colon \mathbb{R} \to \mathbb{R}$ is required to be differentiable.

Mutability is used only to determine which weights should be updated after back propagation. In particular, M is not used in forward propagation, and we often omit M in situations where it plays no role, including forward propagation.

Remark 2.2 The operations $\mathcal{M}$ and $\mathcal{P}$ are called *multiset* and *powerset*. They both form a monad on the category **Set** of sets and functions. In general, they are defined on a set I as:

$$\begin{aligned}\mathcal{P}(I) &= \{S \mid S \subseteq I\}\\ \mathcal{M}(I) &= \{\varphi\colon I \to \mathbb{R} \mid \mathrm{supp}(\varphi) \text{ is finite}\},\end{aligned}$$

where $\mathrm{supp}(\varphi) = \{i \in I \mid \varphi(i) \neq 0\}$ is the support of φ. Such a function φ can also be written as formal sum:

$$\varphi \equiv r_1\,|i_1\rangle + \cdots + r_m\,|i_m\rangle \quad \text{where} \quad \begin{cases}\mathrm{supp}(\varphi) = \{i_1, \ldots, i_m\} \subseteq I\\ r_k = \varphi(i_k) \in \mathbb{R}.\end{cases}$$

This explains why such an element $\varphi \in \mathcal{M}(I)$ is sometimes called a *multiset* on I: it counts elements $i_k \in I$ with multiplicity $r_k = \varphi(i_k) \in \mathbb{R}$.

In this paper we shall use these monads $\mathcal{P}$ and $\mathcal{M}$ exclusively on natural numbers, as finite sets; in that case $\mathcal{M}(n) = \mathbb{R}^n$, as used above.

We shall not really use that $\mathcal{P}$ and $\mathcal{M}$ are monads, except for the following construction: each function $T\colon I \to \mathcal{M}(J)$ has a 'Kleisli' or 'linear' extension $T_*\colon \mathcal{M}(I) \to \mathcal{M}(J)$ given by:

$$T_*(\varphi)(j) = \sum_{i \in I} T(i)(j) \cdot \varphi(i). \tag{1}$$

The transistion map T in a layer $n \Rightarrow k$ is the *linear* part of the associated function $\mathbb{R}^n \to \mathbb{R}^k$, and the activation function α is the *non-linear* part. This linear role of T is emphasised by using this linear extension T_*.

Notice that if $T(i)(j) = 0$, then the input from node i does not contribute to the outcome. Hence this corresponds to not having a connection $i \to j$ in the layer. When it comes to updating, we have to distinguish between a weight being 0 because there is no connection — so that it remains 0 — and weights that happen to be zero at some point in time, but may become non-zero after an update. This is done via the mask function M.

Definition 2.3 Let $\langle T, M, \alpha\rangle$ be a layer $n \Rightarrow k$ as in Definition 2.1. It gives rise to a (differentiable) function $[\![\, T, \alpha \,]\!]\colon \mathbb{R}^n \to \mathbb{R}^k$ in the following manner.

$$[\![\, T, \alpha \,]\!](\boldsymbol{x}) \coloneqq \boldsymbol{\alpha}\big(T_*(\boldsymbol{x}, 1)\big). \tag{2}$$

Notice that we use notation $\boldsymbol{x} \in \mathbb{R}^n$ to indicate a vector of reals $x_i \in \mathbb{R}$. Similarly, the notation $\boldsymbol{\alpha}$ is used to apply $\alpha\colon \mathbb{R} \to \mathbb{R}$ coordinate-wise to $T_*(\boldsymbol{x}, 1) \in \mathbb{R}^k$, where T_* is defined in (1). The additional input 1 in $T_*(\boldsymbol{x}, 1)$ is used to handle biases, as will be illustrated in the example below.

The function $[\![\, T, \alpha \,]\!]\colon \mathbb{R}^n \to \mathbb{R}^k$ expresses (forward) state transformation. Some-

times we use alternative notation $\gg$ for state transformation, defined as:

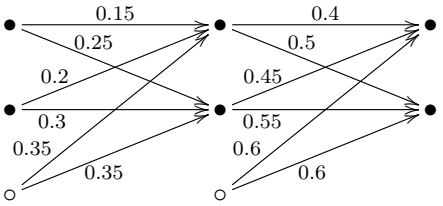
$$(T, \alpha) \gg \boldsymbol{x} \;:=\; [\![T, \alpha]\!](\boldsymbol{x}).$$

This notation is especially suggestive in combination with loss transformation $\ll$, working backwards.

The interpretation function $[\![T, \alpha]\!]$ performs what is often called *forward propagation*. We will refer to vectors $\boldsymbol{x} \in \mathbb{R}^n$ as *states*; they describe the numerical values associated with n nodes at a particular stage in a neural network. We can then also say that forward propagation involves *state transformation*—a layer $n \Rightarrow k$ transforms states in $\mathbb{R}^n$ to states in $\mathbb{R}^k$.

The following example [4] illustrates how the interpretation function works.

Example 2.4 Consider the following neural network with two layers.



(3)

We shall describe this network as two layers:

$$2 \xrightarrow{\langle T,M,\sigma \rangle} 2 \xrightarrow{\langle S,M,\sigma \rangle} 2 \qquad \text{where} \qquad \begin{cases} M(i) = 2 = \{0, 1\} \\ \sigma(z) = \frac{1}{1+e^{-z}} \end{cases}$$

In this network all connections are mutable, as indicated via the function M which sends each $i \in 2$ to the whole subset $M(i) = 2 \subseteq 2$. The activation function is the so-called sigmoid function σ, for both layers, given by $\sigma(z) = 1/(1+e^{-z})$.

The two transition functions T, S have type $3 \to \mathcal{M}(2)$. Their definition is given by the labels on the arrows in the network (3):

$$\begin{aligned} T(0) &= 0.15\,|0\rangle + 0.25\,|1\rangle \\ T(1) &= 0.2\,|0\rangle + 0.3\,|1\rangle \\ T(2) &= 0.35\,|0\rangle + 0.35\,|1\rangle \end{aligned} \qquad \begin{aligned} S(0) &= 0.4\,|0\rangle + 0.5\,|1\rangle \\ S(1) &= 0.45\,|0\rangle + 0.55\,|1\rangle \\ S(2) &= 0.6\,|0\rangle + 0.6\,|1\rangle\,. \end{aligned}$$

Alternatively, one may see T, S as matrices:

$$T = \begin{pmatrix} 0.15 & 0.2 & 0.35 \\ 0.25 & 0.3 & 0.35 \end{pmatrix} \qquad S = \begin{pmatrix} 0.4 & 0.45 & 0.6 \\ 0.5 & 0.55 & 0.6 \end{pmatrix}$$

[4] The example is taken from Matt Mazur's blog, at https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.

We thus get, according to (2):

$$\begin{aligned}
[\![T, \sigma]\!](x_0, x_1) &= \langle\, \sigma\big(T_*(x_0, x_1, 1)(0)\big),\ \sigma\big(T_*(x_0, x_1, 1)(1)\big)\,\rangle \\
&= \langle\, \sigma\big(T(0)(0) \cdot x_0 + T(1)(0) \cdot x_1 + T(2)(0) \cdot 1\big), \\
&\qquad \sigma\big(T(0)(1) \cdot x_0 + T(1)(1) \cdot x_1 + T(2)(1) \cdot 1\big)\,\rangle \\
&= \langle\, \sigma\big(0.15 \cdot x_0 + 0.2 \cdot x_1 + 0.35\big), \sigma\big(0.25 \cdot x_0 + 0.3 \cdot x_1 + 0.35\big)\,\rangle \\
[\![S, \sigma]\!](y_0, y_1) &= \langle\, \sigma\big(0.4 \cdot y_0 + 0.45 \cdot y_1 + 0.6\big), \sigma\big(0.5 \cdot y_0 + 0.55 \cdot y_1 + 0.6\big)\,\rangle.
\end{aligned}$$

We see how the bias is described via the arrows out of the 'open' nodes $\circ$ in (3) and is added in the appropriate manner to the outcome, via the value '1' on the right-hand-side in (2).

The network transforms an initial state $\langle 0.05, 0.1\rangle \in \mathbb{R}^2$ first into[5] :

$$[\![T, \sigma]\!](0.05, 0.1) = \langle\, \sigma(0.3775), \sigma(0.3925)\,\rangle = \langle\, 0.59326999, 0.59688438\,\rangle$$

Subsequently it yields as final state:

$$[\![S, \sigma]\!](0.59327, 0.59688) = \langle\, \sigma(1.10591), \sigma(1.22492)\,\rangle = \langle\, 0.75136507, 0.77292847\,\rangle.$$

We write **NN** for the category of neural networks, as in [1]. Its objects are natural numbers $n \in \mathbb{N}$, corresponding to n nodes. A morphism $n \to k$ in **NN** is a sequence of layers $n \Rightarrow \cdots \Rightarrow k$, forming a neural network. Composition in **NN** is given by concatenation of sequences; a (tagged) empty sequence is used as identity map for each object n.

Next, we write **RF** for the category of real multivariate differentiable functions: objects are natural numbers and morphisms $n \to k$ are differentiable functions $\mathbb{R}^n \to \mathbb{R}^k$.

Proposition 2.5 *Forward state transformation (propagation) yields a functor* $\mathbf{NN} \to \mathbf{RF}$*, which is the identity on objects. A morphism* $n \to k$ *in* **NN***, given by a sequence of layers* $\langle \ell_1, \cdots, \ell_m \rangle$*, is sent to the composite* $[\![\, \ell_m \,]\!] \circ \cdots \circ [\![\, \ell_1 \,]\!] \colon \mathbb{R}^n \to \mathbb{R}^k$*, with the understanding that an empty sequence* $\langle\rangle \colon n \to n$ *in* **NN** *gets sent to the identity function* $\mathbb{R}^n \to \mathbb{R}^n$*. This yields a functor by construction.* □

In line with this description we shall interpret a morphism $N = \langle \ell_1, \ldots, \ell_m \rangle \colon n \to k$ in the category **NN** as a function $[\![N]\!] = [\![\ell_m]\!] \circ \cdots \circ [\![\ell_1]\!] \colon \mathbb{R}^n \to \mathbb{R}^k$. We also write $N \gg x$ for $[\![N]\!](x)$.

3 Backward loss transformations

In the theory of neural networks one uses 'loss' functions to evaluate how much the outcome of a computation differs from a certain 'target'. A common choice is the

[5] The calculations here, and in Example 4.5 have been done with simple Python code, using the `numpy` library.

following. Given outcomes $\boldsymbol{y} \in \mathbb{R}^k$ and a target $\boldsymbol{t} \in \mathbb{R}^k$ one takes as loss:

$$\tfrac{1}{2} \sum\nolimits_i (y_i - t_i)^2$$

Here we abstract away from the precise form of such computations and use a function L for loss. In fact, we incorporate the target $\boldsymbol{t}$ in the loss function, so that for the above example we can give L the type $L \colon \mathbb{R}^k \to \mathbb{R}$, with definition:

$$\boldsymbol{y} \models L \;\coloneqq\; L(\boldsymbol{y})$$

The validity notation $\models$ emerges from the view that vectors $\boldsymbol{y} \in \mathbb{R}^k$ are states (of type k), and loss functions $L \colon \mathbb{R}^k \to \mathbb{R}$ are predicates (of type k). The notation $\boldsymbol{y} \models L$ then expresses the value of the loss L in the state $\boldsymbol{y}$.

We now come to backward transformation of loss along a layer. We ignore mutability because it does not play a role.

Definition 3.1 Let $(T, \alpha) \colon n \Rightarrow k$ be a single layer. Each loss function $L \colon \mathbb{R}^k \to \mathbb{R}$ on the codomain k of this layer can be transformed into a loss function $(T, \alpha) \ll L \colon \mathbb{R}^n \to \mathbb{R}$ on the domain n via:

$$(T, \alpha) \ll L \;\coloneqq\; L \circ [\![T, \alpha]\!] \;:\; \mathbb{R}^n \to \mathbb{R}^k \to \mathbb{R}.$$

For a morphism $N = \langle \ell_1, \ldots, \ell_m \rangle \colon n \to k$ in the category **NN** of neural networks we define:

$$N \ll L \;\coloneqq\; \ell_1 \ll \cdots (\ell_n \ll L) = L \circ [\![\ell_m]\!] \circ \cdots \circ [\![\ell_1]\!] = L \circ [\![N]\!].$$

We can now formulate a familiar property for validity and transformations, see *e.g.* [4,6].

Lemma 3.2 *For any neural network $N \colon n \to k$ in* **NN***, any loss function $L : \mathbb{R}^k \to \mathbb{R}$ and any state $\boldsymbol{x} \in \mathbb{R}^n$, one has:*

$$N \gg \boldsymbol{x} \models L \quad = \quad \boldsymbol{x} \models N \ll L. \tag{4}$$

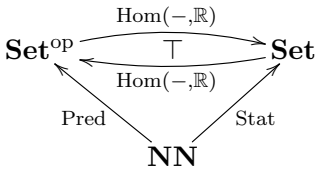
Proof By the definition of these notations:

$$\begin{aligned} N \gg \boldsymbol{x} \models L &= L\big(N \gg \boldsymbol{x}\big) \\ &= \big(L \circ [\![N]\!]\big)(\boldsymbol{x}) \\ &= \big(N \ll L\big)(\boldsymbol{x}) \\ &= \boldsymbol{x} \models N \ll L. \end{aligned}$$

□

Many forms of state and predicate transformation can be described in the form of a 'state-and-effect triangle', where 'effect' is used as alternative name for 'predicate', see [6]. Here this takes the following form.

Theorem 3.3 *There are state and predicate functors* Stat *and* Pred *in a triangle:*



given by:

$$\mathrm{Pred}(n) = \mathbb{R}^{\mathbb{R}^n} \qquad \mathrm{Stat}(n) = \mathbb{R}^n$$
$$\mathrm{Pred}(N) = N \ll (-) = (-) \circ [\![N]\!] \qquad \mathrm{Stat}(N) = N \gg (-) = [\![N]\!] \circ (-).$$

□

The above triangle commutes in one direction: $\mathrm{Hom}(-,\mathbb{R}) \circ \mathrm{Stat} = \mathrm{Pred}$. In order to obtain commutation in the other direction one typically restricts the category **Set** to an appropriate subcategory of algebraic structures. For instance, in probabilistic computation, states form convex sets and predicates form effect modules, see *e.g.* [4,5]. In the present situation with neural nets it remains to be investigated which algebraic structures are relevant. That is not so clear in the current general set up, for instance because we impose no restrictions on the loss functions that we use.

4 Back propagation

In the setting of neural networks, back propagation is a key step to perform an update of (the linear part of) a layer. Here we shall give an abstract description of such updates, in terms of a loss function L as used in the previous section. In fact, we assume that what is commonly called the learning rate η is also incorporated in L.

Let $\langle T, M, \alpha \rangle \colon n \to k$ be a layer. Given an input state $\boldsymbol{a} \in \mathbb{R}^n$ and an (differentiable) loss predicate $L \colon \mathbb{R}^k \to \mathbb{R}$ we will define a *gradient*

$$\boldsymbol{\nabla}_{(\boldsymbol{a},L)}(T) \qquad \text{and use it to change } T \text{ into} \qquad T - M \odot \boldsymbol{\nabla}_{(\boldsymbol{a},L)}(T),$$

where the mutability map $M \colon n \to \mathcal{P}(k)$ is used as $k \times n$ Boolean matrix (with 0's and 1's only), and where $\odot$ is the Hadamard product, given by elementwise multiplication. It ensures that only mutable connections are updated.

Definition 4.1 In the situation just described, the gradient can be given as:

$$\boldsymbol{\nabla}_{(\boldsymbol{a},L)}(T) \coloneqq \Big(\tfrac{\partial}{\partial X} \big((X, \alpha) \gg \boldsymbol{a} \models L \big) \Big)(T). \tag{5}$$

We have introduced a new bound variable X, to clearly indicate the derivative that we are interested in. The type of X is the same as T, namely a $k \times (n+1)$ matrix.

In order to compute this gradient, we recall that the derivative of a (differentiable) function $f\colon \mathbb{R}^n \to \mathbb{R}^m$ is the $m \times n$ 'Jacobian' matrix of partial derivatives:

$$f' = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

Lemma 4.2 *In the situation of Definition 4.1,*

(i) *The gradient $\boldsymbol{\nabla}_{(\boldsymbol{a},L)}(T)$ can be calculated as:*

$$\boldsymbol{\nabla}_{(\boldsymbol{a},L)}(T) = \boldsymbol{s} \cdot (\boldsymbol{a}, 1)^{\mathsf{T}} \qquad \text{where} \qquad s_j = L'((T,\alpha) \gg \boldsymbol{a})_j \cdot \alpha'(T_*(\boldsymbol{a},1)_j).$$

(The superscript T in $(-)^{\mathsf{T}}$ is for 'matrix transpose', and is unrelated to the transition map T.)

(ii) *In the special case where α is the sigmoid function σ, the vector $\boldsymbol{s}$ in point (i) is a Hadamard product:*

$$\boldsymbol{s} = L'(\boldsymbol{b}) \odot \boldsymbol{b} \odot (1 - \boldsymbol{b}) \qquad \text{where} \qquad \boldsymbol{b} = (T,\sigma) \gg \boldsymbol{a}.$$

Proof The chain rule for multivariate functions gives a product of matrices:

$$\boldsymbol{\nabla}_{(\boldsymbol{a},L)}(T) = L'\big((T,\alpha) \gg \boldsymbol{a}\big) \cdot \boldsymbol{\alpha}'\big(T_*(\boldsymbol{a},1)\big) \cdot \Big(\tfrac{\partial}{\partial X} X_*(\boldsymbol{a},1)\Big)(T). \tag{6}$$

We elaborate the three parts one-by-one.

- The derivative of the loss function $L\colon \mathbb{R}^k \to \mathbb{R}$ is given by its partial derivatives, written as $L'\colon \mathbb{R}^k \to \mathbb{R}^k$. Thus, the first part $L'\big((T,\alpha) \gg \boldsymbol{a}\big)$ of (6) is in $\mathbb{R}^k$.
- The derivative of the coordinate-wise application $\boldsymbol{\alpha}\colon \mathbb{R}^k \to \mathbb{R}^k$ of $\alpha\colon \mathbb{R} \to \mathbb{R}$, applied to the sequence $T_*(\boldsymbol{a},1) \in \mathbb{R}^k$ consists of the $k \times k$ diagonal matrix with entries $\alpha'(T_*(\boldsymbol{a},1)_j)$ at position j,j. We shall write this diagonal as a vector $\langle \alpha'(T_*(\boldsymbol{a},1)_j) \rangle \in \mathbb{R}^k$.

  The product of the first two factors in (6) can thus be written as a Hadamard (coordinatewise) product $\odot$:

$$L'\big((T,\alpha) \gg \boldsymbol{a}\big) \odot \langle \alpha'(T_*(\boldsymbol{a},1)_j) \rangle.$$

- For the third part in (6) we notice that $X \mapsto X_*(\boldsymbol{a},1)$ is a function $\mathbb{R}^{k\times(n+1)} \to \mathbb{R}^k$. The j^{th} row of its Jacobian consists of the $k \times (n+1)$ matrix with $\boldsymbol{a}, 1$ at row j and zeros everywhere else. Indeed, the j^{th} coordinate $X_*(\boldsymbol{a},1)_j$ is given by:

$$X_*(\boldsymbol{a},1)_j = X_{j1}a_1 + \cdots + X_{jn}a_n + X_{j(n+1)}.$$

  Taking its derivative with respect to the variables X_{ji} yields the $k \times (n+1)$

matrix:

$$\begin{pmatrix} \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ a_1 & \cdots & a_n & 1 \\ \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \end{pmatrix} \quad \leftarrow \text{ row } j$$

Thus, $\big(\frac{\partial}{\partial X} X_*(\boldsymbol{a}, 1)\big)(T)$ consists of k-many of such matrices stacked on top of each other.

(i) Writing $s_j = L'((T,\alpha) \gg \boldsymbol{a})_j \cdot \alpha'(T_*(\boldsymbol{a},1)_j)$ we can put the previous three bullets together and write the gradient $\boldsymbol{\nabla}_{(\boldsymbol{a},L)}(T)$ as an outer product:

$$\begin{pmatrix} a_1 s_1 & \cdots & a_n s_1 & s_1 \\ \vdots & & \vdots & \\ a_1 s_k & \cdots & a_n s_k & s_k \end{pmatrix} = \begin{pmatrix} s_1 \\ \vdots \\ s_k \end{pmatrix} \cdot \begin{pmatrix} a_1 & \cdots & a_n & 1 \end{pmatrix} = \boldsymbol{s} \cdot (\boldsymbol{a}, 1)^{\mathsf{T}}.$$

(ii) Directly from (i) since $\sigma' = \sigma(1-\sigma)$. □

Next we are interested in gradients of multiple layers.

Proposition 4.3 *Consider two consecutive layers $m \xrightarrow{(S,\beta)} n \xrightarrow{(T,\alpha)} k$, with initial state $\boldsymbol{a} \in \mathbb{R}^m$ and loss function $L\colon \mathbb{R}^k \to \mathbb{R}$. The gradient for updating S is:*

$$\begin{aligned} \Big(\tfrac{\partial}{\partial X}\big((T,\alpha) \gg (X,\beta) \gg \boldsymbol{a} \models L\big)\Big)(S) &\overset{(4)}{=} \Big(\tfrac{\partial}{\partial X}\big((X,\beta) \gg \boldsymbol{a} \models (T,\alpha) \ll L\big)\Big)(S) \\ &= \boldsymbol{\nabla}_{(a,(T,\alpha)\ll L)}(S). \end{aligned}$$

The derivative of the transformed loss $(T,\alpha) \ll L$ is by the chain rule:

$$\big((T,\alpha) \ll L\big)'(\boldsymbol{y}) = \Big(L'\big([\![T,\alpha]\!](\boldsymbol{y})\big) \odot \boldsymbol{\alpha}'\big(T_*(\boldsymbol{y},1)\big)\Big) \cdot [T], \tag{7}$$

where $[T]$ is the $k \times n$ matrix obtained from the $k \times (n+1)$ matrix T by omitting the last column.

More generally, for appropriately typed neural nets N, M,

$$\Big(\tfrac{\partial}{\partial X}\big(N \gg (X,\alpha) \gg M \gg \boldsymbol{a} \models L\big)\Big)(S) = \boldsymbol{\nabla}_{(M \gg a, N \ll L)}(S).$$

Proof The first equation in the above proposition obviously holds. We concentrate on the second equation (7):

$$\begin{aligned} \big((T,\alpha) \ll L\big)'(\boldsymbol{y}) &= \big(L \circ [\![T,\alpha]\!]\big)'(\boldsymbol{y}) \\ &= L'\big([\![T,\alpha]\!](\boldsymbol{y})\big) \cdot [\![T,\alpha]\!]'(\boldsymbol{y}) \\ &= \Big(L'\big([\![T,\alpha]\!](\boldsymbol{y})\big) \odot \boldsymbol{\alpha}'\big(T_*(\boldsymbol{y},1)\big)\Big) \cdot T'_*(\boldsymbol{y},1) \\ &= \Big(L'\big([\![T,\alpha]\!](\boldsymbol{y})\big) \odot \boldsymbol{\alpha}'\big(T_*(\boldsymbol{y},1)\big)\Big) \cdot [T]. \end{aligned}$$

We still need to prove $T'_*(\boldsymbol{y}, 1) = [T]$, where $[T]$ is obtained from T by dropping the last column. The function $T_*(-, 1)$ has type $\mathbb{R}^n \to \mathbb{R}^k$, so the derivative $T'_*(\boldsymbol{y}, 1)$ is a $k \times n$ matrix with entry at i, j given by:

$$\frac{\partial T_*(\boldsymbol{y},1)_i}{\partial y_j} = \frac{\partial(T_{i1}y_1 + \cdots + T_{in}y_n + T_{i(n+1)})}{\partial y_j} = T_{ij}.$$

Together these T_{ij}, for $1 \leq i \leq k$ and $1 \leq j \leq n$, form the $k \times n$ matrix $[T]$. □

Remark 4.4 Equation (7) reveals an important point: for actual computation of backpropagation we are not so much interested in *loss* transformation, but in *erosion* transformation, where we introduce the word 'erosion' as name for the derivative L' of the loss function L.

For this erosion transformation we introduce new notation $\lll$. Let $(T, \alpha)\colon n \Rightarrow k$ be a single layer, and let $E\colon \mathbb{R}^k \to \mathbb{R}^k$ be a 'erosion' function. We transform it into another erosion function $(T, \alpha) \lll E\colon \mathbb{R}^n \to \mathbb{R}^n$, by following (7):

$$\big((T, \alpha) \lll E\big)(\boldsymbol{x}) \coloneqq \Big(E\big(\llbracket T, \alpha \rrbracket(\boldsymbol{x})\big) \odot \boldsymbol{\alpha}'\big(T_*(\boldsymbol{x}, 1)\big)\Big) \cdot [T]. \tag{8}$$

By construction we have:

$$\big((T, \alpha) \ll L\big)' = (T, \alpha) \lll L'. \tag{9}$$

Conceptually, we consider loss transformation more fundamental than erosion tranformation, because loss transformation gives rise to the 'triangle' situation in Theorem 3.3. In addition, erosion transformation can be expressed via derivatives and loss transformation, as the above equation (9) shows.

In the obvious way we can extend $\lll$ in (8) from single to multiple layers (neural networks). In case α is the sigmoid function σ, the right-hand-side of (8) simplifies to:

$$\big((T, \sigma) \lll E\big)(\boldsymbol{x}) = \Big(E(\boldsymbol{y}) \odot \boldsymbol{y} \odot (1 - \boldsymbol{y})\Big) \cdot [T] \qquad \text{where} \qquad \boldsymbol{y} = (T, \sigma) \gg \boldsymbol{x}. \tag{10}$$

We illustrate back propagation for the earlier example.

Example 4.5 We continue Example 2.4 and compute the relevant gradients for updating the transition maps/matrices T, S in the neural network (3) with two layers:

$$2 \xrightarrow{(T,\sigma)} 2 \xrightarrow{(S,\sigma)} 2$$

We shall write input, intermediary, and final states, as computed in Example 2.4, respectively as:

$$\begin{aligned} \boldsymbol{a} &= \langle 0.05, 0.1 \rangle \\ \boldsymbol{b} &= (T, \sigma) \gg \boldsymbol{a} = \langle 0.59326999, 0.59688438 \rangle \\ \boldsymbol{c} &= (S, \sigma) \gg \boldsymbol{b} = \langle 0.75136507, 0.77292847 \rangle. \end{aligned}$$

The target in this example is $\langle 0.01, 0.99\rangle \in \mathbb{R}^2$, so that the loss function $L\colon \mathbb{R}^2 \to \mathbb{R}$ and its 'erosion' derivative $E = L'\colon \mathbb{R}^2 \to \mathbb{R}^2$ are:

$$L(\boldsymbol{x}) = \tfrac{1}{2}\eta\big((x_1 - 0.01)^2 + (x_2 - 0.99)^2\big) \qquad E(\boldsymbol{x}) = \eta\langle x_1 - 0.01, x_2 - 0.99\rangle.$$

The learning rate η is set to 0.5.

The updating of the transition matrices T, S works in backward direction. By Lemma 4.2 we get as gradient:

$$\begin{aligned}\boldsymbol{\nabla}_{((T,\sigma)\gg\boldsymbol{a},L)}(S) &= \boldsymbol{s}\cdot(\boldsymbol{b},1)^{\mathsf{T}} \qquad \text{where } \boldsymbol{s} = E(\boldsymbol{c})\odot\boldsymbol{c}\odot(1-\boldsymbol{c})\\ &= \begin{pmatrix} 0.08216704 & 0.08266763 & 0.13849856\\ -0.02260254 & -0.02274024 & -0.03809824\end{pmatrix}.\end{aligned}$$

Hence the updated last transition function / matrix S is:

$$S - \boldsymbol{s}\cdot(\boldsymbol{b},1)^{\mathsf{T}} = \begin{pmatrix} 0.35891648 & 0.40866619 & 0.53075072\\ 0.51130127 & 0.56137012 & 0.61904912\end{pmatrix}$$

Our next aim is to update the preceding, first transition function / matrix T.

$$\begin{aligned}\boldsymbol{\nabla}_{(\boldsymbol{a},(S,\sigma)\ll L)}(T) &= \boldsymbol{t}\cdot(\boldsymbol{a},1)^{\mathsf{T}}\\ &\text{where}\quad \boldsymbol{t} = ((T,\sigma)\lll E)(\boldsymbol{b})\odot\boldsymbol{b}\odot(1-\boldsymbol{b})\\ &\overset{(10)}{=} \Big(\big(E(\boldsymbol{c})\odot\boldsymbol{c}\odot(1-\boldsymbol{c})\big)\cdot[T]\Big)\odot\boldsymbol{b}\odot(1-\boldsymbol{b})\\ &= \big(\boldsymbol{s}\cdot[T]\big)\odot\boldsymbol{b}\odot(1-\boldsymbol{b})\\ &= \begin{pmatrix} 0.00043857 & 0.00087714 & 0.00877135\\ 0.00049771 & 0.00099543 & 0.00995425\end{pmatrix}.\end{aligned}$$

The updated first matrix of the neural network is then:

$$T - \boldsymbol{t}\cdot(\boldsymbol{c},1)^{\mathsf{T}} = \begin{pmatrix} 0.14978072 & 0.19956143 & 0.34561432\\ 0.24975114 & 0.29950229 & 0.34502287\end{pmatrix}$$

This corresponds to the numbers given in Mazur's blog mentioned in footnote 4, except that there the biases are not updated. This example illustrates that backpropagation can be done in a recursive manner, since the values $\boldsymbol{s}$ in the first step are re-used in $\boldsymbol{t}$ in the second step.

5 Functoriality of backpropagation

In a recent paper [1] a categorical analysis of neural networks is given. Its main result is compositionality of backpropagation, via a description of backpropagation as a functor. In this section we first give a description of the functoriality of backpropagation in the current framework, and then give a comparison with [1].

We write **SL** for the category of 'states and losses'.

- The objects of **SL** are triples $(n, \boldsymbol{a}, L)$, where $\boldsymbol{a} \in \mathbb{R}^n$ is a state of type n and $L\colon \mathbb{R}^n \to \mathbb{R}$ is a (differentiable) loss function of the same type n.
- A morphism $N\colon (n, \boldsymbol{a}, L) \to (k, \boldsymbol{b}, K)$ is a neural network $N\colon n \to k$, in the category **NN**, such that both: $\boldsymbol{b} = N \gg \boldsymbol{a}$ and $K = N \ll L$.

There is an obvious forgetful functor $\mathcal{U}\colon \mathbf{SL} \to \mathbf{NN}$ given by $\mathcal{U}(n, \boldsymbol{a}, L) = n$ and $\mathcal{U}(N) = N$.

Definition 5.1 Define *backprop* $\mathcal{B}\colon \mathbf{SL} \to \mathbf{NN}$ in the following way. On objects, we simply take $\mathcal{G}(n, \boldsymbol{a}, L) = n$. Next, let $N = \langle \ell_1, \ldots, \ell_m \rangle$ be a morphism $(n, \boldsymbol{a}, L) \to (k, \boldsymbol{b}, K)$ in **SL**, where $\ell_i = (T_i, \alpha_i, M_i)$. We write:

- $\boldsymbol{a}_0 \coloneqq \boldsymbol{a}$ and $\boldsymbol{a}_{i+1} \coloneqq \ell_i \gg \boldsymbol{a}_i$; this gives a list of states $\langle \boldsymbol{a}_0, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_m \rangle$ with $\boldsymbol{a}_m = \boldsymbol{b}$, by assumption;
- $K_m \coloneqq K$ and $K_{i-1} \coloneqq \ell_i \ll K_i$; this gives a list of loss functions $\langle K_0, \ldots, K_m \rangle$ with $K_0 = L$.

Then $\mathcal{B}(N)\colon n \to k$ is defined as a list of layers, of the same length m as N, with components:

$$\mathcal{B}(N)_i \coloneqq \langle T_i - M_i \odot \boldsymbol{\nabla}_{(\boldsymbol{a}_{i-1}, K_i)}(T_i),\ \alpha_i,\ M_i \rangle.$$

(Recall, M_i is a Boolean 'mask' matrix that takes care of mutability, and $\odot$ is the Hadamard product.)

Theorem 5.2 *Backprop* $\mathcal{B}\colon \mathbf{SL} \to \mathbf{NN}$ *is a functor.*

Proof This is 'immediate', but writing out the details involves a bit of book keeping. Let $(n, \boldsymbol{a}, L) \xrightarrow{N} (k, \boldsymbol{b}, K) \xrightarrow{P} (l, \boldsymbol{c}, F)$ be (composible) morphisms in **SL**, where $N = \langle \ell_1, \ldots, \ell_u \rangle$ and $P = \langle p_1, \ldots, p_v \rangle$. We write $\ell_i = \langle T_i^N, \alpha_i^N, M_i^N \rangle$ and similarly $p_j = \langle T_j^P, \alpha_j^P, M_j^P \rangle$. The procedures in the two bullets in Definition 5.1 yield for the maps N and K separately:

- $\langle \boldsymbol{a}_0, \ldots, \boldsymbol{a}_u \rangle$ and $\langle \boldsymbol{b}_0, \ldots, \boldsymbol{b}_v \rangle$ where $\boldsymbol{a}_0 = \boldsymbol{a}, \boldsymbol{a}_{i+1} = \ell_i \gg \boldsymbol{a}_i$ and $\boldsymbol{b}_0 = \boldsymbol{b}, \boldsymbol{b}_{j+1} = p_j \gg \boldsymbol{b}_i$; we have $\boldsymbol{b}_0 = \boldsymbol{b} = N \gg \boldsymbol{a} = N \gg \boldsymbol{a}_0 = \boldsymbol{a}_u$;
- $\langle K_0, \ldots, K_u \rangle$ and $\langle F_0, \ldots, F_v \rangle$ with $K_u = K, K_{i-1} = \ell_i \gg K_i$ and $F_v = F, F_{j-1} = p_j \gg F_j$; then $K_u = D = P \ll F = P \ll F_v = F_0$.

From the perspective of the composite sequence $\langle \ell_1, \ldots, \ell_u, p_1, \ldots, p_v \rangle$ we can go through the same process and obtain sequences $\langle \boldsymbol{a}'_0, \ldots, \boldsymbol{a}'_{u+v} \rangle$ and $\langle F'_0, \ldots, F'_{u+v} \rangle$ with:

$$\boldsymbol{a}'_i = \begin{cases} \boldsymbol{a}_i & \text{if } i \leq p \\ \boldsymbol{b}_{i-p} & \text{otherwise} \end{cases} \qquad F'_j = \begin{cases} K_j & \text{if } j \leq p \\ F_{j-p} & \text{otherwise.} \end{cases}$$

We can now describe the components of the updated network $\mathcal{B}(P \circ N)$. For

$1 \leq i \leq u$ and $1 \leq j \leq v$,

$$\begin{aligned}
\mathcal{B}(P \circ N)_i &= \langle T_i^N - M_i^N \odot \boldsymbol{\nabla}_{(\boldsymbol{a}'_{i-1}, F'_i)}(T_i^N),\, \alpha_i^N,\, M_i^N \rangle \\
&= \langle T_i^N - M_i^N \odot \boldsymbol{\nabla}_{(\boldsymbol{a}_{i-1}, K_i)}(T_i^N),\, \alpha_i^N,\, M_i^N \rangle \\
&= \mathcal{B}(N)_i \\
&= \big(\mathcal{B}(P) \circ \mathcal{B}(N)\big)_i \\
\mathcal{B}(P \circ N)_{u+j} &= \langle T_j^P - M_j^P \odot \boldsymbol{\nabla}_{(\boldsymbol{a}'_{u+j-1}, F'_{u+j})}(T_j^P),\, \alpha_j^P,\, M_j^P \rangle \\
&= \langle T_j^P - M_j^P \odot \boldsymbol{\nabla}_{(\boldsymbol{b}_{j-1}, F_j)}(T_j^P),\, \alpha_j^P,\, M_j^P \rangle \\
&= \mathcal{B}(P)_j \\
&= \big(\mathcal{B}(K) \circ \mathcal{B}(N)\big)_{p+j}.
\end{aligned}$$

□

We conclude this section with a comparison to [1], where it was first shown that backpropagation is functorial. The approach in [1] is both more abstract and more concrete than ours.

(i) Here, a layer $(T, \alpha)\colon n \Rightarrow k$ of a neural network consists of linear part $T\colon n+1 \to \mathcal{M}(k)$ and a non-linear part $\alpha\colon \mathbb{R} \to \mathbb{R}$. We ignore the mutability matrix M for a moment. As shown in Definition 2.3, the layer (T, α) gives rise to an interpretation function $[\![T, \alpha]\!]\colon \mathbb{R}^n \to \mathbb{R}^k$ that performs forward state transformation $(T, \alpha) \gg (-)$. In [1] there is no such concrete description of a layer. Instead, the paper works with 'parametrised' functions $P \times \mathbb{R}^n \to \mathbb{R}^k$. Our approach fits in this framework by taking the set of linear parts $P = \mathcal{M}(k)^{n+1} = (\mathbb{R}^k)^{n+1}$ as parameter set. These parametrised functions are organised in a category Para, which is shown to be symmetric monoidal.

(ii) The comparison of the outcome of a state transformation by a network $n \Rightarrow k$ and a target $\boldsymbol{t} \in \mathbb{R}^k$ is captured here abstractly via a loss function $L\colon \mathbb{R}^k \to \mathbb{R}$. This more general perspective allows us to define loss transformation $N \ll L$ along a network N. We have thus developed a view on neural network computation, with forward and backward transformations, that is in line with standard approached to (categorical) program semantics. It gives rise to the pattern of a state-and-effect triangle in Theorem 3.3. Moreover, we show that there is an associated 'erosion transformation' function, that is suitable related to loss transformation via derivatives, see (9).

In the formalism of [1] backward computation also plays a role, via a request function 'r', of type $P \times \mathbb{R}^n \times \mathbb{R}^k \to \mathbb{R}^n$, for a network $n \Rightarrow k$. It corresponds to our erosion transformation (8), roughly as: $r(\ell, \boldsymbol{a}, \boldsymbol{b}) = (\ell \lll L'_{\boldsymbol{b}})(\boldsymbol{a})$, where $L'_{\boldsymbol{b}}$ is the derivative of the loss function $L_{\boldsymbol{b}}$ associated with the 'target' $\boldsymbol{b}$.

(iii) Here we have concentrated on the sequential structure. In [1], parallel composition is also taken into account in the form of symmetric monoidal structure. For us, such additional structure is left as future work.

6 Conclusions

In this paper, we have examined neural networks as programs in a state-and-effect framework. In particular, we have characterized the application of a neural network to an input as a kind of state transformation via Kleisli composition and backpropagation of loss along the network as a kind of predicate transformation on losses. We also observed that the compositionality of backpropagation corresponds to the functoriality of a mapping between a category of states-and-effects to the category of neural networks.

For the sake of illustrating this perspective on neural networks, we have deliberately chosen a simple subclass of the known network architectures and built a category of multilayer perceptron (MLPs). However, we believe it is possible to develop a richer categorical structure capable of capturing a much wider variety of network architectures. This may be the focus of future work.

We also considered a single training scheme: backpropagation paired with stochastic gradient descent (with a fixed learning rate). We are interested in modeling other kinds of neural network training categorically.

As mentioned in the discussion following Theorem 3.3, there is typically a category of algebraic structures in the upper right vertex of the state-and-effect triangle which we have not determined yet.

Acknowledgment

The first author (BJ) acknowledges support from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement n^{o} 320571. The second author (DS) is supported by the JST ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603).

References

[1] Fong, B., D. Spivak and R. Tuyéras, *Backprop as functor: A compositional perspective on supervised learning* (2019), lICS 2019, to appear; see arxiv.org/abs/1711.10455.

[2] Ghica, D., K. Muroya, S. Chung, V. Darvariu and R. Rowe, *A functional perspective on machine learning via programmable induction and abduction*, in: J. Gallagher and M. Sulzmann, editors, *Functional and Logic Programming*, number 10818 in Lect. Notes Comp. Sci. (2018), pp. 84–98.

[3] Goodfellow, I., Y. Bengio and A. Courville, "Deep Learning," MIT Press, 2016, http://www.deeplearningbook.org.

[4] Jacobs, B., *New directions in categorical logic, for classical, probabilistic and quantum logic*, Logical Methods in Comp. Sci. **11(3)** (2015), see https://lmcs.episciences.org/1600.

[5] Jacobs, B., *From probability monads to commutative effectuses*, Journ. of Logical and Algebraic Methods in Programming **94** (2017), pp. 200–237.

[6] Jacobs, B., *A recipe for state and effect triangles*, Logical Methods in Comp. Sci. **13(2)** (2017), see https://lmcs.episciences.org/3660.

[7] Jacobs, B. and F. Zanasi, *The logical essentials of Bayesian reasoning*, in: *Probabilistic Programming* (book chapter, to appear in 2019), see arxiv.org/abs/1804.01193.

[8] Rumelhart, D. E., G. E. Hinton and R. J. Williams, *Learning representations by back-propagating errors*, Nature **323** (1986), pp. 533–536.