Smooth Nondeterministic Arrows

Zenna Tavares
Massachusetts Institute of Technology
zenna@mit.edu

Armando Solar Lezama
Massachusetts Institute of Technology
asolar@csail.mit.edu

Abstract

We present Smooth Nondeterministic Arrows (SNA): a differentiable programming language with refinement types. Smooth arrows combine modern functional programming concepts with favourable properties of deep neural networks. We build upon the framework of arrows, limiting types to real-valued arrays to preserve differentiability, but extending the type system to support non-determinism. The result is a language which supports reuse of computation and parameters within a model; is comprehensible and composable by humans and hence accomodates (partially specified) domain specific knowledge; and naturally generalises most neural network models.

1 Introduction

Recently, artificial neural networks (ANNs) have advanced the state of the art in computer vision, speech recognition and a variety of subfields of machine learning and probabilistic inference. ANNs are computable functions, i.e., programs, and hence these feats can be viewed as achievements in program synthesis. However, when viewed as a *language* for expressing functions, ANNs possess few of the constructs found in contemporary programming languages. The consequence of this is that ANNs (i) often possess far more parameters than is necessary [4, 16], (ii) are not easily interpretable or composable by humans, and (iii) lack compositionality and modularity, hence do not readily support the reuse of parameters or computations within or between models.

Two questions motivate our contribution. First, what are the salient properties of ANNs that enable them to tackle problems beyond the domain of logic-based program synthesis methods (e.g. [17, 18]). Second, can these shortcomings of ANNs be alleviated by adopting concepts found in modern programming languages. To this end, we present sub-differentiable, array-typed and parametrically polymorphic arrows; or in short, *Smooth Nondeterministic Arrows* (SNA).

Arrows [6] are a concept developed in functional programming, but are an abstract view of computation in their own right. Arrows generalise monads and are strongly related to Freyd categories [1]. For our purposes however, it suffices to view arrows as analogous to circuits; multi-input, multi-output functional units that can be wired together arbitrarily, i.e., composed in more exotic ways than basic function composition.

SNA is restricted to integer and sub-differentiable array transformations. On the other hand, SNA extends the arrow formalism with two types of nondeterminism. Refinement-type [14] based nondeterminism allows more properties of arrays (e.g. dimensionality, shape, values) to be reasoned about by the type system, leading to simpler and more general programs that are less tedious to write. Parametric nondeterminism semantically distinguishes parameters from inputs, and gives parameterised arrows first class status through a set of combinators and laws. Together, these properties allow SNA to express complex function spaces with few parameters; naturally generalise the majority of existing neural network architectures; and support problems which require both learning and domain-specific, human-specified knowledge (we build a ray-tracing arrow for inverse-rendering in figure 1).

2 Complementary Program Representations

The representations, methods and measures of correctness used in learning and inference differ significantly from those used in programming languages and formal methods. To synthesise these two areas, we first identity those properties which appear critical to different program representations.

Although the effectiveness of modern artificial neural networks out-paces any conclusive explanation thereof, a number of formal and informal proposals have been made [2, 13, 10, 3]. ANNs learn to represent concepts *distributed* as a pattern over a large number of cells [5], which in many situations can express exponentially more concepts than non-distributed ones [2]. Modern ANNs are *deep*: the number of paths, and hence ways to re-use different components of a network, grows exponentially with depth. They are also *sub-differentiable*; exact derivatives for ANNs are obtained through back-propagation [15]. Particularly in high dimensions, gradient (and higher order derivative) information is crucial for effective learning by optimisation.

Programming languages vary wildly. Nevertheless, a number of concepts have emerged as crucial to developing complex programs, mostly independent of language. *Recursion* enables the reuse of computation, and allows the size of a program to be independent of its input. Modern languages structure a computation into a unit - for instance a procedure, function or method - which can be *referred to* by name or address. This simple property dramatically increases the complexity of computations that can be expressed with a given number of bits. Many languages also express *nondeterminism*, most commonly in the form of type based polymorphism. Declarative languages elevate nondeterminism to a central role. This allows the programmer to specify only what they know, and leaves an inference procedure to determine the unknowns.

3 Smooth Nondeterminstic Arrows

Arrows [6, 12] represent computations that are function-like, in that they take inputs and produce some output. Arrows have more structure than functions, and can be composed in more exotic ways than simple function composition. Building complex programs using arrows involves wiring simpler arrows together, hence arrows bear resemblance to electronic circuits, data-flow languages [7], hardware description languages, and even neural networks.

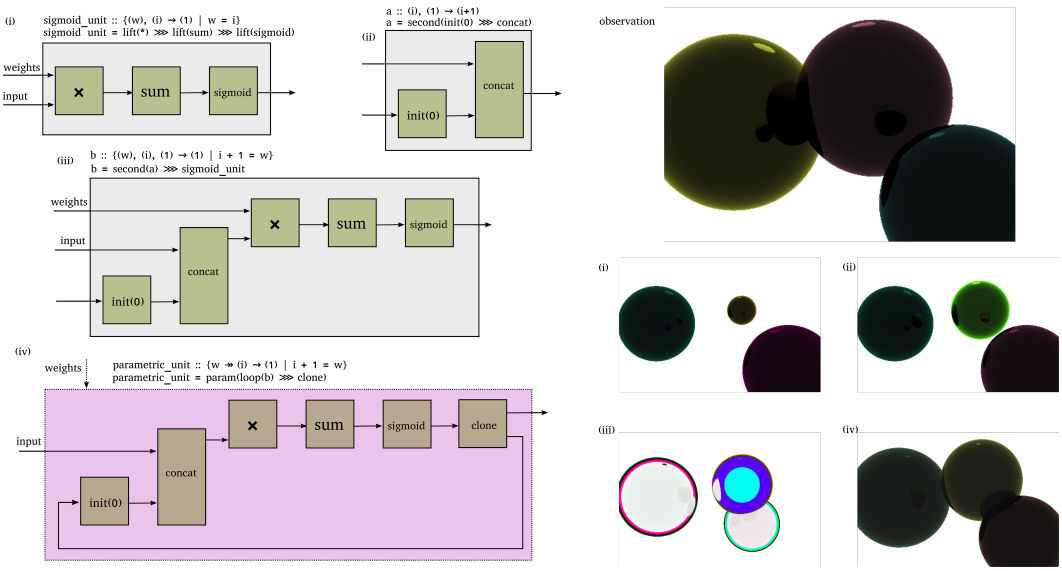


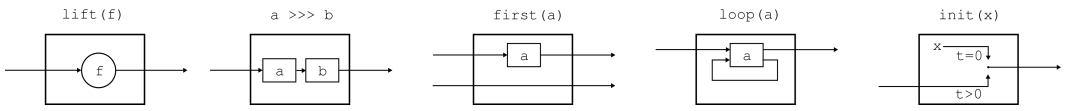
Figure 1: *Left*. Building a recurrent sigmoid arrow set. At each intermediate stage, the combinators used and resulting arrow and type are shown. *Right*: Application of arrows to inverse-rendering [8]. Arrows can combine learning with domain-specific knowledge (e.g. ray-tracing). (i) to (iv) visualise stages of optimisation over scene (3D geometry, lighting, colours, etc), such that ray-traced scene is close to observed image at *top*.

Arrows are best explained by example. A sigmoid unit can be easily described as an arrow. This arrow should take as input a vector of weights and a vector of inputs, and return their dot product filtered through a sigmoid non-linearity. Using smooth arrows, we could write this unit as follows:

$$lift(*) \ggg lift(sum) \ggg lift(sigmoid)$$

$lift$ here is an *arrow combinator*; all complex arrows are made using combinators and all combinators can be derived from the primitive combinators shown in figure 2. $lift$ converts a function (e.g. $*$ which does pointwise multiplication of two vectors) into the corresponding arrow. The composition combinator $\ggg$ pipes the output of one arrow into the input of another. Note that construction of this arrow was *point free*; we made no reference to the underlying values being passed through the arrows. Point free programming simplifies equational reasoning about programs.

Recursion is modeled with cyclic arrows whose input depends on output from a previous time. Recursive arrows are constructed using the $loop$ combinator [11]. $loop$ wires the last output of an arrow into its last input. Liu et al [9] observed that the looping laws and and combinators in [11] were in fact too general. In particular they are not *causal*; $loop$ allows us to construct arrows whose input depends on output at the same time. To restrict the class of arrows to be causal, any use of $loop$ must be accompanied by $init$. $init$ is a combinator which takes a value x, and builds an arrow whose output is initially x (regardless of input), but is thereafter its input (i.e., it then behaves like the identity arrow). $init$ enforces a temporal order of computation and hence causality. Figure 1 (i) and (ii) show how to convert the sigmoid unit into a recurrent network, where the output (initially set to 0 using $init$) is connected, using $loop$, to its input.



$$lift_{\alpha,\beta} : (\alpha \to \beta) \to (\alpha \rightsquigarrow \beta)$$
$$\ggg_{\alpha,\beta,\gamma} : (\alpha \rightsquigarrow \beta) \times (\beta \rightsquigarrow \gamma) \to (\alpha \rightsquigarrow \gamma) \quad loop_{\alpha,\beta,\gamma} : (\alpha \times \gamma \rightsquigarrow \beta \times \gamma) \to (\alpha \rightsquigarrow \beta)$$
$$first_{\alpha,\beta,\gamma} : (\alpha \rightsquigarrow \beta) \to (\alpha \times \gamma \rightsquigarrow \beta \times \gamma) \qquad init_{\alpha} : \alpha \to (\alpha \rightsquigarrow \alpha)$$

Figure 2: Arrows are created with combinators. $\alpha \times \beta \to \gamma$ and $\alpha \times \beta \rightsquigarrow \gamma$ respectively denote a function and arrow from two input values of type α and β respectively, to value of type γ. $lift$ converts a function f into a corresponding arrow. The composition arrow $\ggg$ wires the output of a into the input of b. $first$ (and $second$ as used in figure 1) returns an arrow with an additional input below (above) the existing ones. Recursion is modeled with $loop$; causality is enforced with $init$.

3.1 Types based nondeterminism

Types ensure that arrows are well-formed; almost all well-typed smooth arrows will execute without error. Additionally, types which contain parameters provide a form of nondeterminism, which especially when combined with *arrow sets*, allow us to write very abstract, modular programs.

Array Types. Real valued multidimensional arrays are most basic data type of SNA. Arrays are flat (not nested) and homogeneous; they contain scalars all of the same type. An array type represents a set of arrays. Notationally we use tuples to describe an array shape; for instance, $(10, 20)$ refers to all matrices of 10 rows and 20 columns.

A type can be *parametrically polymorphic* - which means it contains variables - with respect to any combination of (i) the type of scalar it holds, (ii) the number of dimensions it has, (iii) its shape - the number of elements in each dimension - or (iv) the values it contains. The type (m, n) is the set of matrices of m rows and n columns; it is fixed in dimensionality (two) but parametric in shape. If m and n are both unconstrained, (m, n) is the set of all matrices. Variables with the same symbol are constrained to hold the same value; hence (m, m) is the set of all square matrices. Type parameters can also be constrained, hence the type $\{(m, n) \mid m > n\}$ refers to the set of all matrices with more rows than columns.

The type system of SNA, like all type systems, is conservative. For example, the output of the sigmoid arrow is bounded between 0 and 1, but this is not expressed in its type. However, for array computations the type is often sufficiently informative to eliminate incorrect programs and resolve values which would otherwise be specified manually. With weaker type systems, building complex array programs often leads to complicated and tedious data reshaping. In SNA, the types of arrows such as $reshape$ and $transpose$ contain the required information for these details to be left unspecified, and solved automatically using solver based type inference.

Arrow Types. An arrow type represents a set of possible arrows. It contains the type for each input and output and additional constraints on any variables. For instance the type of the concatenation arrow $concatarr$ which concatenates two vectors is written: $concatarr :: (a), (b) \rightsquigarrow (a + b)$ This means that $concatarr$ is an arrow with two inputs, which are vectors of size a and b respectively. The output is vector which has $a + b$ elements, as we expect. We could have also used $concatarr :: \{(a), (b) \rightsquigarrow (c) \mid c = a + b\}$, i.e., introduced a variable c and asserted $c = a + b$.

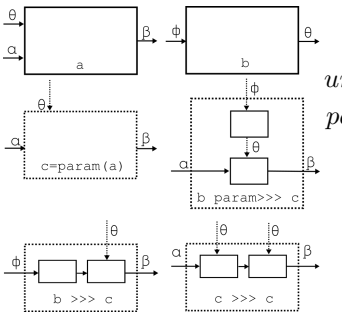
$permute :: x \equiv (x_i, ..., x_n), [p_i, ..., p_n] \rightsquigarrow (x[p_i], ..., x[p_n])$ permutes the dimensions of its first input (e.g. if x has shape $(3, 5, 7)$, $permute(x, [2, 3, 1])$ has shape $(5, 7, 3)$). $x \equiv (x_i, ..., x_n)$ gives the alias x to the shape of the first input; the square brackets in $[p_i, ..., p_n]$ denote that parameters p_i represent *values* of the second array (not its shape). $x[p_i]$ is the p_ith value of the shape array x, hence the ith output dimension will be $x[p_i]$ elements long. If for example we connect a $permute$ arrow to each input of a matrix multiplication arrow, the type system would determine how to permute the input matrices (by assigning p_i values) such that the multiplication is well defined, if possible.

3.2 Parametric Nondeterminism

Learning is typically formulated as a search over a space of functions defined implicitly through parameters, e.g. as a weight matrix in a neural network. Similarly, probabilistic inference methods search over, sample from, or measure sets of values defined by conditioned random variables. The unifying principle of both cases is *nondeterminism*, i.e. a plurality of ways a system could behave.

SNA formalises the distinction between parameters and normal values with the concept of a nondeterministic arrow (*arrow set*) and additional combinators (figure 3). Arrow sets encapsulate computation and parameters into a modular unit. They compose seamlessly with other arrow sets, as well as normal arrows, without the need to reason about parameters explicitly.

A normal arrow is converted into an arrow set using $param$; this is done in figure 1 (iv) to set the weight inputs to the sigmoid unit as parameters. Sigmoid arrow sets can compose together as if they are normal arrows; the resulting arrow set would simply have more parameter inputs. If we wanted to explicitly control how parameters were shared, for instance to force that two composed sigmoid arrow sets used identical parameters, other arrow set combinator such as $param_{\ggg}$ are required.



$$
\begin{aligned}
param_{\alpha,\beta,\theta} &: (\theta \times \alpha \rightsquigarrow \beta) \to (\theta \twoheadrightarrow \alpha \rightsquigarrow \beta) \\
unparam_{\alpha,\beta,\theta} &: (\theta \twoheadrightarrow \alpha \rightsquigarrow \beta) \to (\theta \times \alpha \rightsquigarrow \beta) \\
param_{\ggg\alpha,\beta,\theta} &: (\phi \rightsquigarrow \theta) \times (\theta \twoheadrightarrow \alpha \rightsquigarrow \beta) \to (\phi \twoheadrightarrow \alpha \rightsquigarrow \beta) \\
\ggg_{\alpha,\beta,\gamma,\theta,\phi} &: (\theta \twoheadrightarrow \alpha \rightsquigarrow \beta) \times (\phi \twoheadrightarrow \beta \rightsquigarrow \gamma) \to (\theta \times \phi \twoheadrightarrow \alpha \rightsquigarrow \gamma) \\
\ggg_{\alpha,\beta,\gamma,\theta} &: (\theta \twoheadrightarrow \alpha \rightsquigarrow \beta) \times (\beta \rightsquigarrow \gamma) \to (\theta \twoheadrightarrow \alpha \rightsquigarrow \gamma) \\
\ggg_{\alpha,\beta,\gamma,\theta} &: (\alpha \rightsquigarrow \beta) \times (\theta \twoheadrightarrow \beta \rightsquigarrow \gamma) \to (\theta \twoheadrightarrow \alpha \rightsquigarrow \gamma) \\
first_{\alpha,\beta,\gamma,\theta} &: (\theta \twoheadrightarrow \alpha \rightsquigarrow \beta) \to (\theta \twoheadrightarrow \alpha \times \gamma \rightsquigarrow \beta \times \gamma) \\
loop_{\alpha,\beta,\gamma,\theta} &: (\theta \twoheadrightarrow \alpha \times \gamma \rightsquigarrow \beta \times \gamma) \to (\theta \twoheadrightarrow \alpha \rightsquigarrow \beta)
\end{aligned}
$$

Figure 3: Arrow set combinators. Parameters are on the left side of $\twoheadrightarrow$. $param$ converts an arrow to an arrow set. $\ggg$, $first$ and $loop$ allow arrow sets to be used like and interchangeably with normal arrows. $param_{\ggg}$ allows composition with an arrow set's parameter inputs.

Summary SNA is an arrow based, differentiable language with an refinement types. It generalises neural network models, and is effective for both composing and learning modular programs.

References

[1] Robert Atkey. What is a categorical model of arrows? *Electronic Notes in Theoretical Computer Science*, 229:19–37, 2011.

[2] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation Learning: A Review and New Perspectives. *Tpami*, (1993):1–30, jun 2013.

[3] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The Loss Surfaces of Multilayer Networks. 38, 2014.

[4] Misha Denil, Babak Shakibi, Laurent Dinh, and N de Freitas. Predicting Parameters in Deep Learning. *Advances in Neural . . .* , pages 1–9, 2013.

[5] Geoffrey E Hinton, James L McClelland, and David E Rumelhart. Distributed representations. *The philosophy of artificial intelligence*, pages 248–280, 1990.

[6] John Hughes. Generalizing monads to arrows. *Science of Computer Programming*, 37(1):67–111, 2000.

[7] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[8] Tejas D. Kulkarni, Will Whitney, Pushmeet Kohli, and Joshua B. Tenenbaum. Deep Convolutional Inverse Graphics Network. pages 1–10, 2015.

[9] Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. *ACM SIGPLAN Notices*, 44(9):35, 2009.

[10] Pankaj Mehta and David J. Schwab. An exact mapping between the Variational Renormalization Group and Deep Learning. *arXiv*, pages 1–7, 2014.

[11] Ross Paterson. A new notation for arrows. *ACM SIGPLAN Notices*, 36(10):229, 2001.

[12] Ross Paterson. Arrows and computation. *The Fun of Programming*, pages 201–222, 2003.

[13] Arnab Paul and Suresh Venkatasubramanian. Why does Deep Learning work? - A perspective from Group Theory. pages 1–14, 2014.

[14] Patrick M. Rondon. Liquid types. *ACM SIGPLAN Notices*, pages 1–265, 2012.

[15] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362. 1986.

[16] Tara N. Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. Low-rank matrix factorization for Deep Neural Network training with high-dimensional output targets. *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013*, pages 6655–6659, 2013.

[17] Armando Solar-Lezama. The sketching approach to program synthesis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5904 LNCS:4–13, 2009.

[18] Saurabh Srivastava, Sumit Gulwani, and JeffreyS. Foster. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):497–518, 2013.