Linear logic and recurrent neural networks

Huiyi Hu, Daniel Murfet

January 29, 2017

1 Introduction

In recent years deep neural networks have proven to be proficient at learning hierarchical vector representations of natural data, including images and language [5]. If we follow Leibniz [2] in believing that reasoning is the algebraic manipulation of such representations or "symbols" then it is natural to look for systems which augment the capacity of neural networks for performing these kinds of manipulations. There are by now numerous proposals for how to achieve this, including neural Turing machines [7], stack-augmented recurrent neural networks [21], differentiable neural computers [1] and others [9, 22, 23, 27]. We investigate a new approach, based on the Curry-Howard isomorphism and linear logic.

The Curry-Howard isomorphism [4] gives a bijection between a prototypical system of formal reasoning (Gentzen's natural deduction) and a prototypical algorithmic system (Church's simply-typed lambda calculus). For this reason the lambda calculus and derivative languages such as LISP have played an important role in the symbolic approach to reasoning in the modern field of artificial intelligence. While these methods may have been overshadowed in recent decades by the rise of the connectionist approach, now called deep learning, it has been argued that a synthesis of the two approaches is necessary to achieve general reasoning in a connectionist system [36]. The main obstacle to this synthesis is the discrepancy between the discrete symbolic nature of natural deduction, or equivalently lambda calculus, and the differentiable nature of neural networks. One way to overcome this obstacle is to augment a neural network not directly with symbolic logic, but rather with a *differentiable model* of logic; and the most natural way to construct such a model is not to work directly with simply-typed lambda calculus but rather with a refinement due to Girard known as linear logic [20] which has a canonical model (called a denotational semantics) in differentiable maps between vector spaces [29, 3, 30].

We demonstrate one concrete realisation of these ideas in the form of a Recurrent Neural Network (RNN) controller [6] augmented with spaces of linear logic programs. At each time step the controller predicts which commands (denotations of linear logic programs) to run and the inputs to run them on, from its hidden state and the current input. We call the resulting system the *Linear Logic Recurrent Neural Network* or LLRNN. This

architecture is inspired by many papers in the neural network literature, most notably the second-order RNN [21] and multiplicative RNN [12].

2 Architecture

2.1 The second-order RNN

As usual we use "weight" as a synonym for variable, or more precisely, the variables which we will vary during gradient descent. We denote by σ the function

$$\sigma : \mathbb{R}^k \longrightarrow \mathbb{R}^k$$
$$\sigma(\mathbf{x})_i = \frac{1}{2}(x_i + |x_i|)$$

and by ζ the *softmax* function

$$\zeta : \mathbb{R}^k \longrightarrow \mathbb{R}^k\,,$$
$$\zeta(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^k e^{x_j}}\,.$$

An RNN is defined by its *evolution equation* which expresses $h^{(t+1)}$ as a function of $h^{(t)}, x^{(t)}$ and its *output equation* which gives the output $y^{(t)}$ as a function of $h^{(t)}$. At time $t \geq 1$ we denote the hidden state by $h^{(t)}$ and the input by $x^{(t)}$.

Definition 2.1. A standard Elman-style RNN [6] is defined by weight matrices H, U, B and the evolution equation

$$h^{(t+1)} = \sigma\big(Hh^{(t)} + Ux^{(t+1)} + B\big) \tag{2.1}$$

where $h^{(0)}$ is some specified initial state. The outputs are defined by

$$y^{(t)} = \sigma\big(W_y h^{(t)} + B_y\big)\,. \tag{2.2}$$

Suppose the input vector space is $\mathscr{I}$ and the hidden state space is $\mathscr{H}$, so that $x^{(t)} \in \mathscr{I}$ and $h^{(t)} \in \mathscr{H}$ for all t. The value of the RNN on a sequence $\mathbf{x} = \big(x^{(1)}, \ldots, x^{(T)}\big) \in \mathscr{I}^{\oplus T}$ is computed by applying (2.1) for $0 \leq t \leq T-1$, accumulating the values $\mathbf{y} = (y^{(1)}, \ldots, y^{(T)})$ from each time step, and finally applying a fully-connected layer and the softmax function τ obtain the output sequence $o^{(t)} = \zeta(W_o y^{(t)} + B_o)$. Each $o^{(t)}$ has the property that its components in our chosen basis add to 1. We view such a vector as a probability distribution over the basis, sampling from which gives the output of the RNN on $\mathbf{x}$.

We consider a generalisation of the second-order RNN [16, 17, 14, 15] and the similar multiplicative RNN [12]. In addition to the hidden-to-hidden matrix H and the input-to-hidden matrix U of the traditional RNN, the second-order RNN learns a matrix V that

maps inputs to linear operators on the hidden state. More precisely, a vector is added to the evolution equation whose ith coordinate is given by the formula

$$\sum_j \sum_k V_i^{jk} h_j^{(t)} x_k^{(t+1)} \tag{2.3}$$

where V is interpreted as a tensor in

$$\mathscr{I}^* \otimes \mathscr{H}^* \otimes \mathscr{H} \cong \mathrm{Hom}_{\mathbb{R}}(\mathscr{I}, \mathrm{End}_{\mathbb{R}}(\mathscr{H}))\,. \tag{2.4}$$

Identifying the tensor V with a linear map from the input space $\mathscr{I}$ to linear operators on $\mathscr{H}$, we have that $V(x^{(t+1)})(h^{(t)})$ is the vector whose ith coordinate is (2.3).

Definition 2.2. The second-order RNN [16, 17] is defined by weights H, U, B, V and

$$h^{(t+1)} = \sigma\big(V(x^{(t+1)})(h^{(t)}) + Hh^{(t)} + Ux^{(t+1)} + B\big)\,, \tag{2.5}$$

with $y^{(t)}, o^{(t)}$ as before.

The problem with second-order RNNs is that they may be difficult to train if the state space is large, since $\dim(\mathrm{End}_{\mathbb{R}}(\mathscr{H})) = \dim(\mathscr{H})^2$. The *multiplicative RNN* is introduced in [12] as a more tractable model. The central idea is the same, but an auxiliary space $\mathscr{K}$ is introduced and the RNN learns three linear maps

$$V : \mathscr{I} \longrightarrow \mathrm{End}_{\mathbb{R}}(\mathscr{K})\,, \quad I : \mathscr{K} \longrightarrow \mathscr{H}\,, \quad J : \mathscr{H} \longrightarrow \mathscr{K}\,. \tag{2.6}$$

Moreover V factors through the subspace of diagonalisable matrices in some chosen basis, so it is defined by $\dim(\mathscr{K})$ free parameters. The additional term in the evolution equation (2.5) is changed to

$$I\big(V(x^{(t+1)})(Jh^{(t)})\big)\,. \tag{2.7}$$

The multiplicative RNN has been applied to character-level text generation [12] and sentiment analysis [11]. This is not to be confused with the *multiplicative integration* RNN of [10] which adds a term $Hh^{(t)} \odot Ux^{(t+1)}$ to the evolution equation.

Remark 2.3. The second-order RNN transforms input symbols into linear operators on its hidden state. Observe that consecutive symbols $x = x^{(t)}$ and $x' = x^{(t+1)}$ in the input become composed operators on the hidden state, since (ignoring the non-linearity)

$$\begin{aligned} V(x')(h^{(t)}) &= V(x')\big(V(x)(h^{(t-1)}) + \cdots\big) \\ &= \big\{V(x') \circ V(x)\big\}(h^{(t-1)}) + \cdots\,. \end{aligned}$$

The relevance of this "compositionality" to NLP is remarked on in [11].

2.2 The linear logic RNN

At each time step the second-order RNN generates from its input a linear operator which is applied to to the hidden state; this operator moreover depends *linearly* on the input given to the RNN. The LLRNN model is similar to the second-order RNN in the sense that it generates at each time step a linear operator which is applied to the hidden state, but dissimilar in that this operator depends *non-linearly* on the input as well as a sequence of other vectors generated from the hidden state.

The non-linear function which computes the operator to be applied to the hidden state at each time step is called the *master algorithm*. It is itself the denotation of a proof in linear logic. The arguments to the master algorithm are:

- A sequence of *command vectors* $p_i^{(t+1)}$ generated via softmax ζ from $h^{(t)}$,
- a sequence of *data vectors* $b_i^{(t+1)}$ generated via σ from $h^{(t)}$,
- an *input vector* $c^{(t+1)}$ generated via σ from $x^{(t+1)}$.

The idea is that the command vectors give distributions over finite-dimensional spaces of linear logic programs, while the data vectors and input vector give the inputs to be fed in some way into the selected programs; the exact details of this "feeding" are specified by the master algorithm itself.

We refer to [18, 30] for reviews of the relevant parts of linear logic, and the proof that proof denotations give smooth maps. Recall that the *types* of intuitionistic linear logic are built from atomic variables via binary connectives $\otimes, \multimap$ (tensor, Hom) and a unary connective ! (called bang). The denotational semantics $[\![-]\!]$ of linear logic in the category $\mathcal{V}$ of vector spaces from [18, §5.1, §5.3] is defined as follows: for a variable x the denotation $[\![x]\!]$ is a chosen finite-dimensional vector space, and

$$\begin{aligned} [\![A \multimap B]\!] &= \mathrm{Hom}_k([\![A]\!], [\![B]\!])\,, \\ [\![A \otimes B]\!] &= [\![A]\!] \otimes [\![B]\!]\,, \\ [\![!A]\!] &= ![\![A]\!]\,, \end{aligned}$$

where $!V$ denotes the universal cocommutative counital coalgebra mapping to V. Here $\mathrm{Hom}_{\mathbb{R}}(W, V)$ denotes the space of linear maps from W to V and $\mathrm{End}_{\mathbb{R}}(V) = \mathrm{Hom}_{\mathbb{R}}(V, V)$.

For any proof π of type A the denotation is a vector $[\![\pi]\!] \in [\![A]\!]$. Moreover to a proof π of type $!A \multimap B$ we naturally associate [18, Definition 5.10] a function $[\![\pi]\!]_{nl} : [\![A]\!] \longrightarrow [\![B]\!]$.

Definition 2.4. The Linear Logic Recurrent Neural Network (LLRNN) is determined by the following data:

- types $P_1, \ldots, P_r$ called the *command types*;
- types $B_1, \ldots, B_s$, called the *data types*;

- a type C, called the *input type*;
- a type A with $[\![A]\!] = \mathscr{H}$, the hidden state space;
- two proofs $\underline{\text{master}}^z$ for $z \in \{\text{in}, \text{out}\}$ of the sequent

$$!P_1, \ldots, !P_r, !B_1, \ldots, !B_s, !C \vdash A \multimap A\,. \tag{2.8}$$

- Finite-dimensional subspaces for $1 \leq i \leq r$ and $1 \leq j \leq s$

$$\mathscr{P}_i \subseteq [\![P_i]\!]\,, \quad \mathscr{B}_j \subseteq [\![B_j]\!]\,, \quad \mathscr{C} \subseteq [\![C]\!] \tag{2.9}$$

  which are spanned by the denotations of linear logic proofs. We refer to the elements of $\mathscr{P}_i$ as *command vectors*, elements of $\mathscr{B}_j$ as *data vectors* and elements of $\mathscr{C}$ as *input vectors*.

From this data we obtain the following functions by restriction:

$$[\![\underline{\text{master}}^z]\!]_{nl} : \mathscr{P}_1 \times \cdots \times \mathscr{P}_r \times \mathscr{B}_1 \times \cdots \times \mathscr{B}_s \times \mathscr{C} \longrightarrow \text{End}_{\mathbb{R}}(\mathscr{H})\,. \tag{2.10}$$

The coupling of the master algorithm to the RNN controller is given by

$$\begin{aligned} p_i^{(t+1)} &= \zeta(W_i^p h^{(t)} + B_i^p) \in \mathscr{P}_i\,, \qquad 1 \leq i \leq r\,, \\ b_j^{(t+1)} &= \sigma(W_j^b h^{(t)} + B_j^b) \in \mathscr{B}_j\,, \qquad 1 \leq j \leq s\,, \\ c^{(t+1)} &= W^c x^{(t+1)} \in \mathscr{C} \end{aligned}$$

and the evolution equation

$$h^{(t+1)} = \sigma\Big(Z_{\text{in}} + Hh^{(t)} + Ux^{(t+1)} + B\Big) + Z_{\text{out}} \tag{2.11}$$

where the new term Z, is for $z \in \{\text{in}, \text{out}\}$,

$$Z_z = [\![\underline{\text{master}}^z]\!]_{nl}\Big(p_1^{(t+1)}, \ldots, p_r^{(t+1)}, b_1^{(t+1)}, \ldots, b_s^{(t+1)}, c^{(t+1)}\Big)(h^{(t)})\,. \tag{2.12}$$

The fundamental fact which makes this model reasonable is the following:

Proposition 2.5. *The functions* $[\![\underline{\text{master}}^z]\!]_{nl}$ *are smooth.*

Proof. This follows from the hypothesis that $\mathscr{P}_i, \mathscr{B}_j$ are generated by denotations of linear logic proofs, and the smoothness of these denotations [30]. □

Moreover, the derivatives of $[\![\underline{\text{master}}^z]\!]_{nl}$ can be computed symbolically using the cut-elimination algorithm of differential linear logic, and it is therefore feasible to implement a general LLRNN in a software package like TensorFlow.

Remark 2.6. It will often be convenient to write $\Gamma = P_1, \ldots, P_r$ for the command types and $\Delta = B_1, \ldots, B_s$ for the data types. A standard notational device in the linear logic literature is to then write $!\Gamma$ for the list prepending $!$ to all the types in the list Γ, that is, $!\Gamma = !P_1, \ldots, !P_r$. With this notation, $\underline{\text{master}}$ is a proof of $!\Gamma, !\Delta, !V \vdash A \multimap A$.

3 Examples of the LLRNN

As defined the LLRNN is a class of models. In this section we explain how to implement various existing augmentations of RNNs, including a subset of the Neural Turing Machine, in the framework of the LLRNN. The guiding intuition from logic is that the complexity of an algorithm is rooted in the kind of *iteration* that it employs; for an exposition in the context of linear logic see [18, §7]. From this point of view, the purpose of augmenting an RNN with linear logic is to provide access to iterators of a complexity "similar" to the function that the neural network is trying to approximate.

The most elementary form of iteration is repetition. The simplest example of this form of iteration in the current context is raising a linear operator to an integer power. This non-linear transformation of an input operator $\alpha \in \mathrm{End}_{\mathbb{R}}(\mathscr{H})$ to the output operator α^n is encoded by a proof $\underline{n}$ in linear logic of the type

$$\mathbf{int}_A = !(A \multimap A) \multimap (A \multimap A)$$

where $[\![A]\!] = \mathscr{H}$. A more interesting kind of iteration takes a pair of linear operators α, β and intertwines them according to a sequence of binary numbers $S \in \{0,1\}^*$. This transformation is encoded by a proof $\underline{S}$ in linear logic of type

$$\mathbf{bint}_A = !(A \multimap A) \multimap \big(!(A \multimap A) \multimap (A \multimap A)\big)\,.$$

We begin this section with a brief exposition on the iterators $\underline{n}$ and $\underline{S}$ before going on to explain how they appear in various special cases of the LLRNN. The iterator $\underline{n}$ is the basic tool in the linear logic implementation of the NTM (Section 3.1) and its generalisations in Section 3.2 and Section 3.3 while the iterator $\underline{S}$ is used in Example 3.5.

The reader may prefer to substitute *program* wherever we write *proof* in what follows, following Curry-Howard [4]. There are two ways of presenting proofs in linear logic: proof trees, and the term calculus of [38] which is a refinement of the lambda calculus, and we will use the latter in the following examples.

For more details on the following examples from linear logic see [30, §3].

Example 3.1. The denotation of a proof π of type $\mathbf{int}_A$ is a function

$$[\![\pi]\!]_{nl} : \mathrm{End}_{\mathbb{R}}([\![A]\!]) \longrightarrow \mathrm{End}_{\mathbb{R}}([\![A]\!])$$

which is a polynomial function of the entries of the input matrix. For each integer $n \geq 0$ there is a proof $\underline{n}$ of type $\mathbf{int}_A$. For example, in the term calculus of [38]

$$\underline{2} = (\lambda q.(\text{copy } q \text{ as } h, g \text{ in } (\lambda z.(\text{derelict}(g)\ (\text{derelict}(h)\ z)))\,.$$

The denotation satisfies $[\![\underline{n}]\!]_{nl}(X) = X^n$.

Example 3.2. The denotation of a proof π of type $\mathbf{bint}_A$ is a polynomial function

$$[\![\pi]\!]_{nl} : \mathrm{End}_{\mathbb{R}}([\![A]\!]) \times \mathrm{End}_{\mathbb{R}}([\![A]\!]) \longrightarrow \mathrm{End}_{\mathbb{R}}([\![A]\!])\,.$$

For every binary sequence $S \in \{0,1\}^*$ there is a corresponding proof $\underline{S}$ of $\mathbf{bint}_A$ [30, §3.2], with for example

$$\underline{001} = (\lambda q.(\lambda p.(\text{copy } q \text{ as } r, s \text{ in } (\lambda z.(\text{derelict}(p)\ (\text{derelict}(s)\ (\text{derelict}(r)\ z)))))))\,.$$

The denotation $[\![\underline{S}]\!]$ sends a pair of matrices X, Y to the product described by S, reading X for 0 and Y for 1, and reading in reverse order. For example,

$$[\![\underline{001}]\!](X,Y) = YXX\,.$$

Example 3.3 (Second-order RNN). With $r = s = 0$ there are no command or data vectors. Take for the input type $C = A \multimap A$ and

$$\mathscr{C} = \mathrm{End}_{\mathbb{R}}(\mathscr{H}) = [\![A \multimap A]\!]\,.$$

so the master algorithm takes a single input, which is a linear operator on the hidden state, and returns such an operator. The only weight matrix involved in the LLRNN beyond the usual RNN is the matrix W^c which maps inputs to linear operators on $\mathscr{H}$.

We choose $\underline{\text{master}}^{\text{in}}$ to be the proof of $!(A \multimap A) \vdash A \multimap A$ which is given by dereliction, so that $[\![\underline{\text{master}}^{\text{in}}]\!]_{nl}(\alpha) = \alpha$ and hence

$$Z_{\text{in}} = W^c(x^{(t+1)})(h^{(t)})\,. \tag{3.1}$$

Thus the LLRNN with these settings is just the second-order RNN.

The most elementary coupling of an RNN to linear logic adds the ability to raise linear operators (generated say from an input symbol, in the manner of the second-order RNN) to a power generated from the hidden state of the RNN.

Example 3.4 (Higher-order RNN). Consider the generalisation of the second-order RNN where the controller predicts at each time step an integer power of the linear operator $W^c(x^{(t+1)})$ to apply to the hidden state. Suppose we allow powers in the range $\{0, \ldots, L\}$. Then at each time step the RNN will generate a distribution $p^{(t+1)}$ over $\{0, \ldots, L\}$ from the current hidden state $h^{(t)}$ by the formula

$$p^{(t+1)} = \zeta(W_p h^{(t)} + B_p)$$

and the evolution equation is

$$Z = \sum_{i=0}^{L} p_i^{(t+1)} \big(W^c(x^{(t+1)})\big)^i (h^{(t)})\,. \tag{3.2}$$

The operation of taking a linear operator and raising it to the nth power is encoded by the proof $\underline{n}$ of type $\mathbf{int}_A$. We can therefore represent the higher-order RNN as a LLRNN,

as follows. There is one command type $\mathbf{int}_A$, no data types, and input type $A \multimap A$. The spaces of command and input vectors are respectively

$$\begin{aligned} \mathscr{P}_1 &= \mathrm{span}(\llbracket \underline{0} \rrbracket, \ldots, \llbracket \underline{L} \rrbracket) \subseteq \llbracket \mathbf{int}_A \rrbracket \,, \\ \mathscr{C} &= \mathrm{End}_{\mathbb{R}}(\mathscr{H}) = \llbracket A \multimap A \rrbracket \,. \end{aligned}$$

We omit $\underline{\text{master}}^{\text{out}}$ and write $\underline{\text{master}}$ for $\underline{\text{master}}^{\text{in}}$ which we take to be the proof of

$$!\mathbf{int}_A,\, !(A \multimap A) \vdash A \multimap A$$

given in the term calculus of [38] by

$$\underline{\text{master}} = (\lambda n.(\lambda a.(\mathrm{derelict}(n)\ a))) \,.$$

This proof has the property that for $n_i \geq 0$ and $\alpha \in \mathrm{End}_{\mathbb{R}}(\mathscr{H})$ and $p_i \in \mathbb{R}$,

$$\llbracket \underline{\text{master}} \rrbracket_{nl}\Big(\sum_i \lambda_i \llbracket \underline{n_i} \rrbracket, \alpha \Big) = \sum_i \lambda_i \alpha^{n_i} \,. \tag{3.3}$$

The coupling of this function to the RNN is via $p_1^{(t+1)}$ which we identify with $p^{(t+1)}$ above, and $c^{(t+1)} = W^c(x^{(t+1)})$. By construction the LLRNN with this configuration reproduces the Z of (3.2) and thus the higher-order RNN.

Since one doesn't need to know linear logic to understand how to raise a linear operator to an integer power, it is natural to wonder to what degree linear logic is actually necessary here. All of our models can be formulated without any mention of linear logic, and indeed we will generally present the functions $\llbracket \underline{\text{master}} \rrbracket_{nl}$ rather than the underlying proof, which we relegate to Appendix A. Nonetheless, the construction of $\underline{\text{master}}$ within linear logic constrains the model and makes conceptually clear the computational ideas involved; these ideas are not necessarily clear from the polynomial algebra that results from applying the denotation functor $\llbracket - \rrbracket$. The strongest example of this point of view is the role of iteration in the LLRNN approach to the Neural Turing Machine and its generalisations.

3.1 Neural Turing Machine

The raising of a linear operator to a power predicted from the hidden state is the crucial architectural feature behind the Neural Turing Machine (NTM) [7] and stack-augmented RNN [21]. The linear operator in the former case rotates the memory state, viewed as a sequence of vectors arranged at the vertices of a regular polygon. The read and write address weightings of the NTM are then controlled by predicting a distribution, at each time step, over the distinct powers of this rotation matrix.

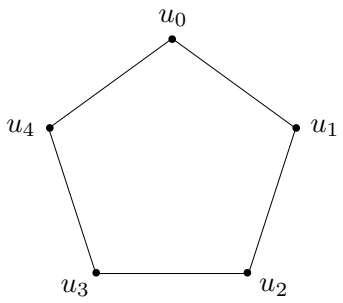
To explain, we introduce the *address space*

$$\mathscr{W} = (\mathbb{Z}/N\mathbb{Z})^{\mathbb{R}} \cong \mathbb{R}\bar{0} \oplus \cdots \oplus \mathbb{R}\overline{N-1}$$

and given a *memory coefficient space* $\mathscr{V}$ the *memory space* is the tensor product

$$\mathscr{S} = \mathscr{W}^* \otimes \mathscr{V} \cong \mathrm{Hom}_{\mathbb{R}}(\mathscr{W}, \mathscr{V})\,.$$

A memory state M is a linear map $M : \mathscr{W} \longrightarrow \mathscr{V}$ which we view as an arrangement of the vectors $u_j = M(\bar{j})$ on the vertices of the regular N-gon:



Sometimes we refer to this structure as a *memory ring*. In the notation of [7] we may take $\mathscr{W} = \mathbb{R}^N$ and $\mathscr{V} = \mathbb{R}^M$ so that elements of $\mathscr{S}$ are identified with $N \times M$ matrices.

The space of memory states has a natural action of the cyclic group of order N, which acts by rotation on $\mathscr{W}$ and therefore on $\mathscr{S}$. The rotated state assigns the vector u_{j+1} to the position j (we index the positions in the clockwise order) with $j+1$ computed modulo N. Said differently, let $R : \mathscr{W} \longrightarrow \mathscr{W}$ be the linear map defined by

$$R(\overline{a}) = \overline{a+1}\,,$$

with dual map $R^* : \mathscr{W}^* \longrightarrow \mathscr{W}^*$. The rotation of the memory state M is the composition $M \circ R$, or what is the same thing, $R^*(M)$. Thus, applying the j-th power of R^* to a memory state has the effect of rotating the state j times around the regular polygon.

The internal state space of the RNN underlying the NTM is decomposed as

$$\mathscr{H} = \mathscr{H}_0 \oplus \mathscr{W} \oplus \mathscr{W}^* \oplus \mathscr{S} \tag{3.4}$$

and the state at time t is written

$$h^{(t)} = (h_0^{(t)}, r^{(t)}, w^{(t)}, M^{(t)}) \in \mathscr{H}\,.$$

The components of this vector are interpreted as the *controller internal state* $h_0^{(t)}$, *read address weights* $r^{(t)}$, *write address weights* $w^{(t)}$ and *memory contents* $M^{(t)}$. For convenience we will refer to the read address weight as the *read address*, even though the vector represents a distribution over the available addresses rather than a specific address. At each time step the RNN generates from its hidden state a distribution s over rotations of

the write address, a distribution q over rotations of the read address, and a vector u to be written to memory, via the formulas

$$\begin{aligned} s^{(t+1)} &= \zeta(W_s h^{(t)} + B_s) \in \mathscr{W}^* \,, \\ q^{(t+1)} &= \zeta(W_q h^{(t)} + B_q) \in \mathscr{W}^* \,, \\ u^{(t+1)} &= \sigma(W_a h^{(t)} + B_a) - \sigma(W_e h^{(t)} + B_e) \in \mathscr{V} \,. \end{aligned}$$

The update equation for the addresses [7, Eq. (8)] and memory [7, Eq. (3),(4)] are

$$w^{(t+1)} = \sum_{i=0}^{N-1} s^{(t+1)}(\bar{i}) \cdot (R^*)^i(w^{(t)}) \,, \tag{3.5}$$

$$r^{(t+1)} = \sum_{i=0}^{N-1} q^{(t+1)}(\bar{i}) \cdot R^i(r^{(t)}) \,, \tag{3.6}$$

$$M^{(t+1)} = M^{(t)} + w^{(t)} \otimes u^{(t+1)} \,. \tag{3.7}$$

Here we write $u^{(t)}$ for the difference $\mathbf{a}_t - \mathbf{e}_t$ of [7, §3.2]. Finally, the evolution equation is

$$h_0^{(t+1)} = \sigma\Big(V\big(M^{(t)}(r^{(t)})\big) + H_0 h_0^{(t)} + U_0 x^{(t+1)} + B_0\Big) \,. \tag{3.8}$$

where $H_0 : \mathscr{H}_0 \longrightarrow \mathscr{H}_0, U_0 : \mathscr{I} \longrightarrow \mathscr{H}_0, V : \mathscr{V} \longrightarrow \mathscr{H}_0$ and $B_0 \in \mathscr{H}_0$ are weight matrices. These equations define the NTM with only location-based addressing, with an RNN controller and memory of size N with a single read and write head.[1]

We now explain how to present this form of the NTM as a special case of the LLRNN. We first give the command, data and input types. Let V, W be types of linear logic with $[\![V]\!] = \mathscr{V}, [\![W]\!] = \mathscr{W}$, and write $W^\vee = W \multimap 1$ so that $[\![W^\vee]\!] = \mathscr{W}^*$. The command types Γ and data types Δ are defined to be respectively (the input type is omitted in this case)

$$\Gamma = \mathbf{int}_W, \mathbf{int}_{W^\vee} \,, \qquad \Delta = W \multimap W, W^\vee \multimap W^\vee, V \,.$$

The master algorithms are the proof of $!\Gamma, !\Delta \vdash A \multimap A$ such that for inputs

$$\begin{gathered} [\![\underline{m}]\!] \in [\![\mathbf{int}_W]\!] \,, \quad [\![\underline{n}]\!] \in [\![\mathbf{int}_{W^\vee}]\!] \\ \alpha \in [\![W \multimap W]\!] = \mathrm{End}_{\mathbb{R}}(\mathscr{W}) \\ \beta \in [\![W^\vee \multimap W^\vee]\!] = \mathrm{End}_{\mathbb{R}}(\mathscr{W}^*) \\ u \in [\![V]\!] = \mathscr{V} \end{gathered}$$

[1] We omit the sharpening step [7, Eq. (9)] for simplicity. The original NTM paper [7] is not specific about how the output $M^{(t)}(r^{(t)})$ of the read heads enters the RNN evolution equation; the form we have given above follows the construction of the differentiable neural computer [1, p. 7]. Note that the term appearing in the equation for $h^{(t+1)}$ is the memory state at time t applied to the output of the read heads at time t. That is, the output of the read heads is incorporated into the hidden state of the RNN at the *next* time step; again, in this architectural choice we are following [1].

we have, for $h = (h_0, r, w, M) \in \mathscr{H}$ decomposed according to (3.4),

$$[\![\underline{\text{master}}^{\text{in}}]\!]_{nl}\big([\![\underline{m}]\!], [\![\underline{n}]\!], \alpha, \beta, u)(h) = \big(V(M(r)), 0, 0, 0\big)\,,$$
$$[\![\underline{\text{master}}^{\text{out}}]\!]_{nl}\big([\![\underline{m}]\!], [\![\underline{n}]\!], \alpha, \beta, u)(h) = \big(0, \alpha^m(r), \beta^n(w), M + w \otimes u\big)\,.$$

For the lambda term, see Appendix A. We take

$$\mathscr{P}_1 = \text{span}([\![\underline{0}]\!], \ldots, [\![\underline{N-1}]\!]) \subseteq [\![\mathbf{int}_W]\!]\,,$$
$$\mathscr{P}_2 = \text{span}([\![\underline{0}]\!], \ldots, [\![\underline{N-1}]\!]) \subseteq [\![\mathbf{int}_{W^\vee}]\!]\,,$$
$$\mathscr{B}_1 = [\![W \multimap W]\!] = \text{End}_{\mathbb{R}}(\mathscr{W})\,,$$
$$\mathscr{B}_2 = [\![W^\vee \multimap W^\vee]\!] = \text{End}_{\mathbb{R}}(\mathscr{W}^*)\,.$$

We restrict the functions $[\![\underline{\text{master}}^z]\!]_{nl}$ to the subset of inputs where $\alpha = R, \beta = R^*$. The coupling of this restricted function to the RNN is via command vectors $p_1^{(t+1)}, p_2^{(t+1)}$ giving distributions over the basis $\{[\![\underline{i}]\!]\}_{i=0}^{N-1}$ of $\mathscr{P}_1, \mathscr{P}_2$ which we identify respectively with $q^{(t+1)}$ (the distribution over powers of R used to manipulate the read address) and $s^{(t+1)}$ (the distribution over powers of R^* used to manipulate the write address).

We assume the weight matrix H of (2.11) is the projection from $\mathscr{H}$ to $\mathscr{H}_0$ followed by the weight H_0 above, and similarly for U, B. Then with the current notation the evolution equation (2.11) of the LLRNN reads

$$\begin{aligned}(h_0^{(t+1)}, r^{(t+1)}, w^{(t+1)}, M^{(t+1)}) = \Big(&\sigma\big(V(M^{(t)}(r^{(t)})) + H_0 h_0^{(t)} + U_0 x^{(t+1)} + B_0\big)\,,\\ &\sum_{i=0}^{N-1}(p_1^{(t+1)})_i R^i(r^{(t)})\,, \sum_{i=0}^{N-1}(p_2^{(t+1)})_i (R^*)^i(w^{(t)})\,,\\ &M^{(t)} + w^{(t)} \otimes u\Big)\,,\end{aligned}$$

which agrees with the equations (3.5) – (3.8).

Example 3.5 (Dihedral NTM). The NTM manipulates its memory state via rotations of the regular N-gon. In this example we study the natural extension which allows access the full symmetry group, the dihedral group, by adding the reflection

$$T : \mathscr{W} \longrightarrow \mathscr{W}\,,$$
$$T(\overline{a}) = \overline{-a}.$$

Note that T and R do not commute. The command and data types are now

$$\Gamma = \mathbf{bint}_W, \mathbf{bint}_{W^\vee}\,,$$
$$\Delta = W \multimap W, W \multimap W, W^\vee \multimap W^\vee, W^\vee \multimap W^\vee, V$$

and $\underline{\text{master}}^{\text{out}}$ is the proof such that for $h \in \mathscr{H}$ and $F, G \in \{0,1\}^*$

$$\begin{aligned}&[\![\underline{\text{master}}^{\text{out}}]\!]_{nl}\big([\![\underline{F}]\!], [\![\underline{G}]\!], \alpha_1, \alpha_2, \beta_1, \beta_2, u)(h)\\ &\qquad = \Big(V(M(r)), [\![F]\!]_{nl}(\alpha_1, \alpha_2)(r), [\![G]\!]_{nl}(\beta_1, \beta_2)(w), M + w \otimes u\Big)\,.\end{aligned}$$

We then fix $\alpha_1 = R, \alpha_2 = T$ and $\beta_1 = R^*, \beta_2 = T^*$.

The Differentiable Neural Computer (DNC) [1] generalises the NTM by allowing for more complicated patterns of memory access based on a probabilistic incidence matrix $L[i,j]$ which records when positions in memory are written in successive time steps. An alternative approach to allowing more complicated memory access patterns is a more thoroughgoing use of iterators; this is more in keeping with the spirit of the LLRNN, and is the perspective we explore in the next section.

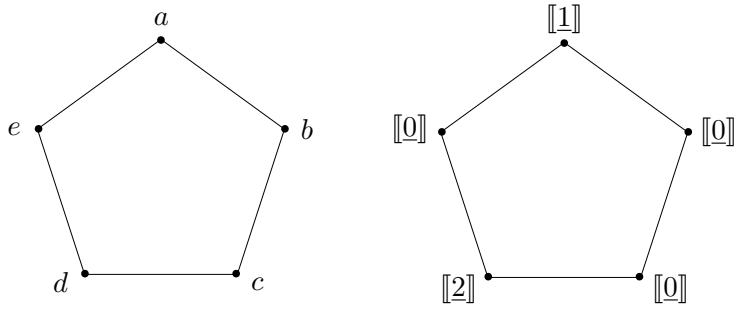
3.2 Pattern NTM

At each time step the NTM predicts distributions $s^{(t)}, q^{(t)}$ over the possible rotations of the write and read addresses. A pattern of memory accesses is a *sequence* of such rotations, and since the "angle" of rotation is represented in the LLRNN as the denotation of a linear logic proof (with $[\![\underline{n}]\!]$ representing a clockwise angle of $\frac{2\pi n}{5}$) the pattern may be stored as a sequence of vectors in $[\![\mathbf{int}_W]\!]$. It is therefore natural to add a memory ring with coefficients in this vector space so the NTM can learn patterns of memory access.

In addition to the memory coefficient space $\mathscr{V}_1 = \mathscr{V}$ from earlier, we now take a finite-dimensional subspace $\mathscr{V}_2 \subseteq [\![\mathbf{int}_W]\!]$. For convenience we keep the same address space for both memory rings. So we have memory spaces for $i \in \{1,2\}$

$$\mathscr{S}_i = \mathscr{W}^* \otimes \mathscr{V}_i \cong \mathrm{Hom}_{\mathbb{R}}(\mathscr{W}, \mathscr{V}_i)\,.$$

An example of a combined state $\mathscr{S} = \mathscr{S}_1 \oplus \mathscr{S}_2$ of these two memory spaces is



We consider a modification of the NTM in which the contents of the second ring control the motion of the read address of the first ring (rather than this being manipulated directly by the RNN controller). To demonstrate the dynamics in an example, suppose that the memory state is as shown at time $t = t_0$, that the read address for the first ring is focused at the zero position at time $t_0 - 1$, and that the read address for the second ring is focused at zero at time t_0 and increases its focus position by one step over each time interval.

Then as time increases from t_0 the repeating sequence

$$[\![\underline{1}]\!], [\![\underline{0}]\!], [\![\underline{0}]\!], [\![\underline{0}]\!], [\![\underline{2}]\!], [\![\underline{0}]\!], \ldots \tag{3.9}$$

will be used to control the read address of the first memory ring. More precisely, the operator applied to the read address $r_1 \in \mathscr{W}$ will be R^n with n varying over the repeating

sequence $1, 0, 0, 2, 0$. Finally, the sequence of vectors delivered to the controller will be

$$a, b, b, b, d, d, e, e, e, b, \dots \ .$$

In more detail, as part of $\underline{\text{master}}$ there will be an evaluation program

$$\begin{array}{c} \underline{\text{eval}} \\ \vdots \\ \mathbf{int}_W, !(W \multimap W) \vdash W \multimap W \end{array}$$

which has the property that

$$[\![\underline{\text{eval}}]\!]\big(\sum_i \lambda_i [\![\underline{n}_i]\!], |\emptyset\rangle_\alpha\big) = \sum_i \lambda_i \alpha^{n_i} \, .$$

The first argument to this evaluation function will be the state $M_2(r_2) = \sum_i \lambda_i [\![\underline{n}_i]\!]$ of the second memory ring. We apply the output to the read address r_1 of the first memory ring and specialise to $\alpha = R$ so that the relevant expression is

$$[\![\underline{\text{eval}}]\!](M_2(r_2), |\emptyset\rangle_R)(r_1) = \sum_i \lambda_i R^{n_i}(r_1) \, .$$

The command and data types are

$$\begin{aligned} \Gamma &= \mathbf{int}_{W^\vee}, \mathbf{int}_W, \mathbf{int}_{W^\vee} \\ \Delta &= W \multimap W, W^\vee \multimap W^\vee, W \multimap W, W^\vee \multimap W^\vee, V, \mathbf{int}_W \, . \end{aligned}$$

The first type in Γ is connected to the write address for the first memory ring, while the second and third types relate respectively to the read and write address of the second memory ring. We do not include an input to the master algorithm for manipulating the read address of the first memory ring since this is purely under the control of the second memory ring. As usual $\underline{\text{master}}^z$ is a proof of $!\Gamma, !\Delta \vdash A \multimap A$.

Given the command and data vectors

$$g = \big([\![\underline{n}_1]\!], [\![\underline{m}_2]\!], [\![\underline{n}_2]\!], \alpha_1, \beta_1, \alpha_2, \beta_2, u_1, u_2\big) \in \bigoplus_{i=1}^{3} \mathscr{P}_i \oplus \bigoplus_{j=1}^{6} \mathscr{B}_j$$

and state vector

$$h = (h_0, r_1, w_1, r_2, w_2, M_1, M_2) \in \mathscr{H} = \mathscr{H}_0 \oplus \mathscr{W} \oplus \mathscr{W}^* \oplus \mathscr{W} \oplus \mathscr{W}^* \oplus \mathscr{S}_1 \oplus \mathscr{S}_2$$

the value of the master functions are

$$[\![\underline{\text{master}}^{\text{in}}]\!]_{nl}(g)(h) = \big(V(M_1(r_1)), 0, 0, 0, 0, 0, 0\big) \, .$$

and

$$\begin{aligned} [\![\underline{\text{master}}^{\text{out}}]\!]_{nl}(g)(h) = \Big(0, [\![\underline{\text{eval}}]\!]\big(M_2(r_2), |\emptyset\rangle_{\alpha_1}\big)(r_1), \beta_1^{n_1}(w_1), \\ \alpha_2^{m_2}(r_2), \beta_2^{n_2}(w_2), M_1 + \beta_1^{n_1}(w_1) \otimes u_1, M_2 + w_2 \otimes u_2\Big) \, . \end{aligned}$$

As above we set $\alpha_1 = \alpha_2 = R$ and $\beta_1 = \beta_2 = R^*$.

3.3 Multiple pattern NTM

In the previous example the second memory ring stores patterns of memory access in the form of sequences of integers, or more precisely, proofs of linear logic of type $\mathbf{int}_W$. More complex patterns (this word being of course a euphemism for *algorithms*) can be encoded using more rings and higher types, in various ways.

In this section we give one example, using a pair of additional rings with coefficients in $[\![\mathbf{int}_W]\!]$ and third additional ring with coefficients in $[\![\mathbf{bint}_W]\!]$. The idea is that the second and third memory rings encode two patterns of memory access, and we allow the controller to switch between the two patterns at any time using the fourth ring.

If at some time the read address of the second and third rings are focused at positions containing $[\![\underline{m}]\!], [\![\underline{n}]\!]$ respectively and the read address of the fourth ring is focused at $[\![\underline{S}]\!]$ for some sequence $S \in \{0,1\}^*$ (notation of Example 3.2) then the read address of the first ring will be acted on by R^p where

$$p = am + bn$$

and a, b are the number of times $0, 1$ appear in S, respectively. Thus whenever we see $[\![\underline{0}]\!]$ on the fourth ring we will follow the pattern on the second ring, and if we see $[\![\underline{1}]\!]$ we will follow the pattern on the third ring. An entry $[\![\underline{00}]\!]$ on the fourth ring will run the pattern on the second ring but with every "angle of rotation" doubled.

The proof that encodes this logic is

$$\begin{array}{c} \underline{\text{feed}} \\ \vdots \\ \mathbf{int}_W, \mathbf{int}_W, \mathbf{bint}_W \vdash \mathbf{int}_W \end{array}$$

with linear lambda term

$$\begin{aligned} \underline{\text{feed}} = (\lambda pqra.(&\text{copy } a \text{ as } a', a'' \text{ in} \\ &(r\ ((\text{promote } y \text{ for } x \text{ in } p)\ a') \\ &\quad ((\text{promote } y' \text{ for } x' \text{ in } q)\ a'')))) \,. \end{aligned}$$

We will not spell out the full model here since it is essentially the same as the one given in the previous section, with the main difference being that the second entry in the tuple defining $[\![\underline{\text{master}}^{\text{out}}]\!]_{nl}(g)(h)$ now looks like

$$[\![\underline{\text{feed}}]\!](M_2(r_2), M_3(r_3), M_4(r_4))_{nl}(R)(r_1)\,. \tag{3.10}$$

To be more concrete, suppose the read address of the fourth ring is sharply focused at a position containing $[\![\underline{S}]\!]$ where S contains a copies of 0 and b copies of 1. If the read addresses of the second and third rings give distributions

$$M_2(r_2) = \sum_i \lambda_i [\![\underline{m}_i]\!]\,, \qquad M_3(r_3) = \sum_j \mu_j [\![\underline{n}_j]\!]$$

then the term in (3.10) is

$$\Big(\big(\sum_i \lambda_i R^{m_i} \big)^a \big(\sum_j \mu_j R^{n_j} \big)^b \Big)(r_1)\,.$$

Clearly this can be written as $\sum_p f_p(\lambda, \mu) R^p(r_1)$ for some polynomials f_p in the variables $\{\lambda_i, \mu_j\}_{i,j}$. The theory of linear logic and its semantics gives a coherent and general way of writing down polynomials, such as $f_p(\lambda, \mu)$, which achieve a given computational idea (in this case, switching between two patterns of memory access) when inserted into the evolution equation of an RNN.

Remark 3.6. A more powerful system would substitute a neural theorem prover along the lines of [26, 28] in place of the libraries of functional programs $\mathscr{P}_i$. At each time step the RNN controller would predict a continuous vector, which when fed into the neural theorem prover as a set of parameters, generates a symbolic program whose denotation is then coupled back into the RNN [27].

Remark 3.7. In functional programming languages like differential λ-calculus [34] and differential linear logic [29] it is possible to differentiate programs with respect to their inputs, even if the programs are higher-order (that is, take functions as input and return them as output). This is a generalisation of automatic differentiation [33] which is widely used to compute derivatives of complicated real-valued functions, for example in the backpropagation algorithms of machine learning packages like TensorFlow [31, §4.1]. The idea is to augment every computation in a code fragment so that derivatives with respect to a chosen variable are computed along with the main result. In TensorFlow this is done by adding special nodes in the dataflow graph.[2] The idea of differential λ-calculus is similar, but more complex [35]. It would be interesting to explore augmenting the dataflow graph of TensorFlow directly with terms of differential linear logic, in a way that generalises the coupling between semantics and RNNs in this paper.

A Lambda terms

In this section we present the linear lambda terms for the master algorithms in some of the earlier examples.

References

[1] Graves, Alex, et al. "Hybrid computing using a neural network with dynamic external memory." Nature 538.7626 (2016): 471-476.

[2] G. Frege, *Begriffschrift, a formula language, modeled upon that of arithmetic, for pure thought*, (1879). An english translation appears in *From Frege to Gödel. A source book in mathematical logic, 1879–1931*, Edited by J. van Heijenoort, Harvard University Press, 1967.t

[3] R. Blute, T. Ehrhard and C. Tasson, *A convenient differential category*, arXiv preprint [arXiv:1006.3140], 2010.

[4] M. H. Sørensen and P. Urzyczyn, *Lectures on the Curry-Howard isomorphism* (Vol. 149), Elsevier, (2006).

[5] Y. LeCun, Y. Bengio and G. Hinton, *Deep learning*, Nature, 521(7553), pp.436–444 (2015).

[2] See the "gradients" method of tensorflow/python/ops/gradients.py in TensorFlow 0.10

[6] J. Elman, *Finding structure in time*, Cognitive science, 14(2):179211, 1990.

[7] A. Graves, G. Wayne and I. Danihelka, *Neural turing machines*, arXiv preprint arXiv:1410.5401 (2014).

[8] A. Joulin and T. Mikolov, *Inferring algorithmic patterns with stack-augmented recurrent nets*, Advances in Neural Information Processing Systems, 2015.

[9] A. Graves, *Hybrid computing using a neural network with dynamic external memory*, Nature 538.7626 (2016): 471–476.

[10] Y. Wu, S. Zhang, Y. Zhang, Y. Bengio and R. R. Salakhutdinov, *On multiplicative integration with recurrent neural networks*, In Advances In Neural Information Processing Systems, pp. 2856-2864. 2016.

[11] O. Irsoy and C. Cardie, *Modeling compositionality with multiplicative recurrent neural networks*, arXiv preprint arXiv:1412.6577 (2014).

[12] I. Sutskever, J. Martens and G. E. Hinton, *Generating text with recurrent neural networks* Proceedings of the 28th International Conference on Machine Learning (ICML-11). 2011.

[13] I. Sutskever, O. Vinyals and Q. V. Le, *Sequence to sequence learning with neural networks*, Advances in neural information processing systems, 2014.

[14] M. W. Goudreau, C. L. Giles, S. T. Chakradhar and D. Chen, *First-order versus second-order single-layer recurrent neural networks*, IEEE Transactions on Neural Networks, 5(3), 511–513, 1994.

[15] C. L. Giles, D. Chen, C. B. Miller, H. H. Chen, G. Z. Sun, Y. C. Lee, *Second-order recurrent neural networks for grammatical inference*, In Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on (Vol. 2, pp. 273-281). IEEE.

[16] C. L. Giles, G. Z. Sun, H. H. Chen, Y. C. Lee, D. Chen, *Higher order recurrent networks and grammatical inference*, In NIPS (pp. 380-387) 1989.

[17] J. B. Pollack, *The induction of dynamical recognizers*, Machine Learning, 7(2-3), 227-252 (1991).

[18] D. Murfet, *Logic and linear algebra: an introduction*, preprint (2014) [arXiv: 1407.2650].

[19] D. Murfet, *On Sweedler's cofree cocommutative coalgebra*, J. Pure and Applied Algebra **219** (2015) 5289–5304.

[20] J.-Y. Girard, *Linear Logic*, Theoretical Computer Science **50** (1987), 1–102.

[21] A. Joulin and T. Mikolov, *Inferring algorithmic patterns with stack-augmented recurrent nets*, Advances in Neural Information Processing Systems, 2015.

[22] E. Grefenstette, et al, *Learning to transduce with unbounded memory*, Advances in Neural Information Processing Systems, 2015.

[23] J. Weston, C. Sumit and B. Antoine, *Memory networks*, preprint (2014) [arXiv:1410.3916].

[24] W. Zaremba, et al., *Learning Simple Algorithms from Examples*, preprint (2015) [arXiv:1511.07275].

[25] P-A. Melliès, *Categorical semantics of linear logic*, in : Interactive models of computation and program behaviour, Panoramas et Synthèses 27, Société Mathématique de France, 2009.

[26] A. A. Alemi, F. Chollet, G. Irving, C. Szegedy and J. Urban, *DeepMath-Deep Sequence Models for Premise Selection*, arXiv preprint arXiv:1606.04442.

[27] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin and D. Tarlow, *DeepCoder: Learning to Write Programs*, arXiv preprint arXiv:1611.01989.

[28] T. Rocktäschel and S. Riedel, *Learning Knowledge Base Inference with Neural Theorem Provers*, In NAACL Workshop on Automated Knowledge Base Construction (AKBC) 2016.

[29] T. Ehrhard, *An introduction to Differential Linear Logic: proof-nets, models and antiderivatives*, [arXiv:1606.01642] (2016).

[30] J. Clift and D. Murfet, *Cofree coalgebras and differential linear logic*, preprint.

[31] Abadi, Martn, et al. *TensorFlow: A system for large-scale machine learning* arXiv preprint arXiv:1605.08695 (2016).

[32] Abadi, Martn, et al. *Tensorflow: Large-scale machine learning on heterogeneous distributed systems*, arXiv preprint arXiv:1603.04467 (2016).

[33] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*, Siam (2008).

[34] T. Ehrhard and L. Regnier, *The differential λ-calculus*, Theoretical Computer Science 309, pp. 1–41, (2003).

[35] O. Manzyuk, *A simply typed λ-calculus of forward automatic differentiation*, In Mathematical Foundations of Programming Semantics Twenty-eighth Annual Conference, pages 259–73, Bath, UK, June 69 2012. [URL].

[36] M. Minsky, *Logical versus analogical or symbolic versus connectionist or neat versus scruffy*, AI magazine, 12(2), 34 (1991).

[37] J.-Y. Girard, Y. Lafont and P. Taylor, *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science 7, Cambridge University Press, 1989.

[38] N. Benton, G. Bierman, V. de Paiva and M. Hyland, *Term assignment for intuitionistic linear logic*, Technical report 262, Computer Laboratory, University of Cambridge, 1992.