Linear logic and recurrent neural networks

Huiyi Hu, Daniel Murfet

April 17, 2017

Contents

1 Introduction

Deep neural networks have proven to be proficient at learning hierarchical vector representations of natural data, including images and language [9]. If we follow Leibniz [6] 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. This is a foundational problem in the field of artificial intelligence which remains far from solved, despite renewed attention in recent years [11, 25, 5, 13, 26, 27, 31, 4, 3]. Arguably one obstacle is that some of the underlying mathematical ideas are still unclear; to help remedy this we investigate a new approach, based on the Curry-Howard correspondence and linear logic.

The Curry-Howard correspondence [8] is 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 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 [40]. 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 [24] which has a canonical model (called a denotational semantics) in differentiable maps between vector spaces [33, 7, 34].

In this paper we realise these ideas in the form of a Recurrent Neural Network (RNN) controller [10] augmented with smooth denotations of linear logic programs. 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 [25], multiplicative RNN [16] and Neural Turing Machine [11]. Conceptually, all of these models augment the basic RNN by generating, at each time step, a *linear operator* from the previous hidden state and the current input. This linear operator is then applied to the previous hidden state to generate the new hidden state. The LLRNN generalises these models by using algorithms from linear logic to generate the linear operator.

2 Background

It is our hope that the paper is accessible both for readers coming from the deep learning community, and for readers coming from the linear logic community. To this end we will make some introductory remarks on both subjects.

2.1 Recurrent Neural Network

We begin with a review of the ordinary Recurrent Neural Network (RNN). A good textbook introduction to deep learning in general and RNNs is [43]. 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 [10] 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 $\mathbf{x}$.

2.2 Second-order Recurrent Neural Network

We consider a generalisation of the second-order RNN [20, 21, 18, 19] and the similar multiplicative RNN [16]. 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 [20, 21] 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.

Remark 2.3. 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 [16] 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 [16] and sentiment analysis [15]. This is not to be confused with the *multiplicative integration* RNN of [14] which adds a term $Hh^{(t)} \odot Ux^{(t+1)}$ to the evolution equation.

Remark 2.4. 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 [15].

2.3 Neural Turing Machine

The Neural Turing Machine (NTM) [11] is an RNN augmented with an additional memory. This memory is read from and written to using "attention" vectors that are manipulated at each time step based on the hidden state. Following our general philosophy, we give in this section a basis-free presentation which emphasises the underlying linear algebra.

The model uses four vector spaces:

- the input space $\mathscr{I}$ and hidden state space $\mathscr{H}_0$ of the underlying RNN, and
- the *address space* $\mathscr{W}$ and *coefficient space* $\mathscr{V}$ of the memory system.

The *space of memory states* is the space of linear maps

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

We think of a basis $e_1, \ldots, e_N$ for $\mathscr{W}$ as "locations" at which may be stored a vector from the coefficient space $\mathscr{V}$. A memory state $M \in \mathscr{S}$ is determined by the vectors $M(e_i) \in \mathscr{V}$, which we view as being stored at the locations e_i. While we will use the basis $\{e_i\}_i$ in the following to illustrate the definitions, it is important to note that the model is defined in a way which avoids explicit reference to any basis.

Definition 2.5. A *read address* is a vector $r \in \mathscr{W}$ and a *write address* is a vector $w \in \mathscr{W}^*$.

Remark 2.6. We may evaluate a memory state M at a read address r to obtain a vector $M(r) \in \mathscr{V}$, and if the read address is *focused* at location i, by which we mean that $r = e_i$, this produces the vector $M(e_i)$ stored at that location. In general a read address specifies a linear combination of the stored vectors. To explain the thinking behind the use of dual vectors as write addresses, consider the canonical isomorphism

$$\Psi : \mathscr{W}^* \otimes \mathscr{V} \longrightarrow \mathrm{Hom}_{\mathbb{R}}(\mathscr{W}, \mathscr{V})$$

under which a tensor $\alpha \otimes v$ with $\alpha \in \mathscr{W}^*, v \in V$ corresponds to the linear map

$$\Psi(\alpha \otimes v)(w) = \alpha(w)v\,.$$

Given a state M of the memory, a location e_i and a vector v to write to this location, the new memory state $M' = M + \Psi(e_i^* \otimes v)$ has the property that

$$M'(e_j) = M(e_j) + \Psi(e_i^* \otimes v)(e_j) = M(e_j) + \delta_{i=j}v\,.$$

Note that in this construction it is the dual vector $w = e_i^*$ which specifies *where to write*, and this explains why we refer to vectors in $\mathscr{W}^*$ as write addresses.

A closer examination of the model in [11] leads us to posit the following:

(i) The RNN controller manipulates read and write addresses by invertible linear transformations, which suggests that we take $\mathscr{W}$ to be a *representation* of a finite group G. At each time step the controller will produce distributions over G which are used to act on the read and write addresses by the action maps

$$G \longrightarrow \mathrm{End}_{\mathbb{R}}(\mathscr{W}), \qquad G \longrightarrow \mathrm{End}_{\mathbb{R}}(\mathscr{W}^*)\,.$$

The second of these maps arises since $\mathscr{W}^*$ is also naturally a representation of G.

(ii) In the erase step (see below) we need to *copy* memory addresses and *multiply* vectors in the coefficient space, so we take $\mathscr{W}$ to be a coalgebra and $\mathscr{V}$ to be an algebra.

The simplest way to realise these constraints is using a finite group $G = \{g_0, g_1, \ldots, g_N\}$ with identity $g_0 = e$ acting by linear transformations on the group algebra

$$\mathscr{W} = \mathbb{R}G = \bigoplus_{g \in G} \mathbb{R}g = \mathbb{R}e \oplus \mathbb{R}g_1 \oplus \cdots \oplus \mathbb{R}g_N$$

which is a vector space with basis G. This is a coalgebra, with comultiplication

$$\Delta : \mathbb{R}G \longrightarrow \mathbb{R}G \otimes \mathbb{R}G\,,$$
$$\Delta(g) = g \otimes g\,.$$

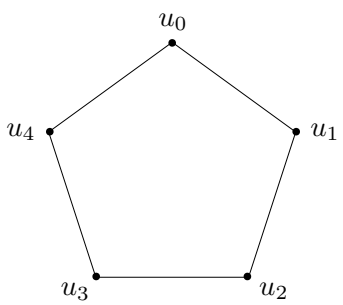
Th group G acts on $\mathbb{R}G$ by the rule

$$G \times \mathbb{R}G \longrightarrow \mathbb{R}G\,,$$
$$(g, \sum_{h \in G} \lambda_h h) \longmapsto \sum_{h \in G} \lambda_h gh\,.$$

In this section we take the group G to be the group of Nth roots of unity

$$G = \{e^{\frac{2\pi i}{N}} \,|\, 0 \leq i \leq N-1\} \cong \mathbb{Z}/N\mathbb{Z}$$

and we write e_i for $e^{\frac{2\pi i}{N}}$ viewed as a basis vector of $\mathscr{W}$. Since we are thinking of these basis vectors as memory locations, we may view a memory state M as an arrangement of the vectors $u_j = M(e_j)$ on the vertices of the regular N-gon in the complex plane



Sometimes we refer to this structure as a *memory ring*.

Finally we take some commutative algebra structure on $\mathscr{V}$. The use of an algebra structure is implicit in [11] and the deep learning literature in general, and enters via the so-called *Hadamard product* $\odot$ which makes $\mathbb{R}^n$ into an algebra via the rule

$$\odot : \mathbb{R}^n \otimes \mathbb{R}^n \longrightarrow \mathbb{R}^n\,,$$
$$\mathbf{x} \odot \mathbf{y} = (x_i y_i)_{i=1}^n\,.$$

We take $\mathscr{V} = \mathbb{R}^M$ with the Hadamard product in what follows. This involves choosing a basis, of course, but we will only refer to the product (a tensor $\odot \in \mathscr{V}^* \otimes \mathscr{V}^* \otimes \mathscr{V}$) and not the basis in what follows.

Definition 2.7 (Erase step). Given a vector $e \in \mathscr{V}$ and a write address $w \in \mathscr{W}^*$ we define an action on the space of memory states $\mathscr{S}$ which we think of as "erasing by e at the address w" as follows. Using

$$\Psi(w \otimes e) \in \mathrm{Hom}_{\mathbb{R}}(\mathscr{W}, \mathscr{V})$$

we may form the composite

$$\mathscr{W} \xrightarrow{\quad\Delta\quad} \mathscr{W} \otimes \mathscr{W} \xrightarrow{M \otimes \Psi(w \otimes e)} \mathscr{V} \otimes \mathscr{V} \xrightarrow{\quad m \quad} \mathscr{V}\,.$$

This is actually the multiplication in a canonical algebra structure on the space $\mathrm{Hom}_{\mathbb{R}}(\mathscr{W}, \mathscr{V})$ which arises from the fact that $\mathscr{W}$ is a coalgebra and $\mathscr{V}$ is an algebra. It is the *convolution product* on the space of memory states. If we write

$$M' = m \circ (M \otimes \Psi(w \otimes e)) \circ \Delta$$

then

$$M'(e_i) = m\,(M(e_i) \otimes \Psi(w \otimes e)(e_i)) = M(e_i) \odot w(e_i)e\,.$$

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

$$\mathscr{H} = \mathscr{H}_0 \oplus \mathscr{W} \oplus \mathscr{W}^* \oplus \mathscr{S} \tag{2.8}$$

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, an operator e to be applied to each coefficient vector in the memory state (this encodes "forgetting") and a vector a to be written to memory, via the formulas

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

The update equation for the addresses [11, Eq. (8)] and memory [11, Eq. (3),(4)] are

$$w^{(t+1)} = \sum_{i=0}^{N-1} s^{(t+1)}(\bar{i}) \cdot (R^*)^i(w^{(t)}) , \tag{2.9}$$

$$r^{(t+1)} = \sum_{i=0}^{N-1} q^{(t+1)}(\bar{i}) \cdot R^i(r^{(t)}) , \tag{2.10}$$

$$M^{(t+1)} = (1_{\mathscr{V}} - e^{(t+1)}) \circ M^{(t)} + w^{(t)} \otimes a^{(t+1)} . \tag{2.11}$$

Finally, the evolution equation is

$$h_0^{(t+1)} = \sigma\Big(M^{(t)}(r^{(t)}) + H_0 h_0^{(t)} + U_0 x^{(t+1)} + B_0\Big) . \tag{2.12}$$

where

$$\begin{aligned} H_0 &: \mathscr{H}_0 \longrightarrow \mathscr{H}_0 , \\ U_0 &: \mathscr{I} \longrightarrow \mathscr{H}_0 , \end{aligned}$$

and $B_0 \in \mathscr{H}_0$ are weight matrices. We assume for simplicity that the memory coefficient space is the same as the hidden state space, that is, $\mathscr{V} = \mathscr{H}$. Following [11, §3.2] we write e for the *erase vector* and a for the *add vector*. These equations define the NTM with location-based addressing, with an RNN controller and memory of size N with a single read and write head.[1]

[1] We omit the sharpening step [11, Eq. (9)] for simplicity. The original NTM paper [11] is not specific

Remark 2.8. To connect the above more explicitly to the notation of [11, Eq. (8)] observe that (2.10) can be rewritten, writing $r[j]$ for the component of $r \in \mathscr{W}$ for $\bar{j}$, as

$$r^{(t+1)}[i] = \sum_j q^{(t+1)}(\bar{j}) r^{(t)}[i-j]\,. \tag{2.13}$$

2.4 Linear Logic

Linear logic is the universal way of describing algorithms which manipulate tensors and linear maps using the basic constructions of linear algebra (such as composition and tensor contraction) and iterations of such constructions. At the level of *syntax* these algorithms may be thought of as programs written in a functional programming language much like the simply-typed lambda calculus. This programming language has a functorial *semantics* denoted here by $[\![-]\!]$ which realises the abstract algorithms as concrete functions.

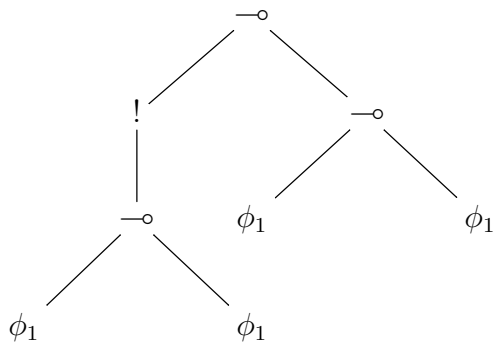
In this section we give a very brief introduction to linear logic; for a longer survey see [22, 34]. We divide the introduction into three parts, covering the *types*, *terms* (meaning programs) and *semantics* of intuitionistic linear logic.

2.4.1 Types

The *types* of intuitionistic linear logic are built from atomic types $\phi_1, \phi_2, \ldots$ via binary connectives $\otimes, \multimap$ (tensor, Hom) and a unary connective ! (called bang). Thus a type may be represented as a syntax tree, where the leaves are labelled by atomic types ϕ_i and the other vertices are labelled by $\otimes, \multimap, !$. For example the type

$$!(\phi_1 \multimap \phi_1) \multimap (\phi_1 \multimap \phi_1)$$

corresponds to the tree



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 [5, 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 [5]. Similarly, the write address used to define $M^{(t+1)}$ is $w^{(t)}$ rather than $w^{(t+1)}$. In this it seems we actually *disagree* with [11, 5]. It makes the formulas cleaner, but maybe it is wrong (TODO).

Given a type A we can construct two new types

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

called respectively the types of *integers* on A and *binary integers* on A, which will play an important role in what follows.

2.4.2 Terms

There is a countable collection of variables $x_1^A, x_2^A, \ldots$ of each type A. Variables are *terms* and a general term is built up from variables by a set *deduction rules* which we will not review here, but which we will try to illuminate with examples. As usual we write $\pi : A$ to indicate that π is a term of type A and present terms as linear lambda terms, following [42]. Some examples of linear lambda terms are, for variables $x : A, y : B$ and $z : !A$,

$$(\lambda x \,.\, x) : A \multimap A\,, \tag{2.14}$$
$$(\lambda x \,.\, (\lambda y \,.\, (y\ x))) : A \multimap ((A \multimap A) \multimap A)\,, \tag{2.15}$$
$$(\lambda z \,.\, \text{derelict}(z)) : !A \multimap A\,. \tag{2.16}$$

Remark 2.9. A term of type $A \multimap B$ should be thought of as a program which takes inputs of type A and returns outputs of type B, under the constraint that the input is used *linearly* (that is, precisely once). For example, the term (2.14) just returns its input, while (2.15) takes two inputs $x : A, y : A \multimap A$ and returns the result $(y\ x)$ of applying the second program to the first. In both cases the inputs are used linearly.

On the other hand a term of type $!A \multimap B$ is a program which takes inputs of type A and returns outputs of type B, with no linearity constraint (so it is like a normal program in the lambda calculus). Usually such programs are constructed by first making copies of the input and then feeding those copies into a program, like those in (2.14),(2.15), which is linear with respect to its inputs. To give such an example we need the *dereliction* operator derelict$(-)$ appearing in (2.16) which takes an input which may be used arbitrarily (z in our example) and returns a restricted form of the input which may only be used linearly.

The simplest nontrivial example is the term (with $q : !(A \multimap A), z : A$)

$$\underline{2} = (\lambda q.(\text{copy } q \text{ as } q', q'' \text{ in } (\lambda z.(\text{derelict}(q'')\ (\text{derelict}(q')\ z))))) \tag{2.17}$$

which is of type

$$\underline{2} : \mathbf{int}_A = !(A \multimap A) \multimap (A \multimap A)\,.$$

Intuitively, this program takes an input $q : !(A \multimap A)$, copies it to form q', q'', and returns

$$(\lambda z.(\text{derelict}(q'')\ (\text{derelict}(q')\ z)))$$

which is itself a program, sending $z : A$ to what is intuitively the result of evaluating q twice on z. The corrrect way to make this precise is as shown above, using the dereliction operator. In more informal language, the overall program $\underline{2}$ has the meaning $q \longmapsto q^2$.

Every term in intuitionistic logic can be interpreted as a linear map between (usually infinite-dimensional) vector spaces, and it is these interpretations (or *denotations*, in the logic terminology) which will be used in the recurrent neural networks.

2.4.3 Semantics

The semantics $[\![-]\!]$ of linear logic in the category $\mathcal{V}$ of vector spaces [22, §5.1, §5.3] assigns to each type A a vector space $[\![A]\!]$ and to each term $\pi : A$ a vector $[\![\pi]\!] \in [\![A]\!]$ in a way which is compositional with respect to the structure of types and terms. We refer to the objects $[\![A]\!], [\![\pi]\!]$ as the *denotations* of A, π respectively.

The semantics depends on a choice, for each atomic type ϕ_i, of a vector space $[\![\phi_i]\!]$. This choice is made freely and once made, determines all denotations, according to the following inductive definitions:

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

where $\mathrm{Hom}_{\mathbb{R}}(W, V)$ is the space of linear maps from W to V (we will also use the notation $\mathrm{End}_{\mathbb{R}}(V) = \mathrm{Hom}_{\mathbb{R}}(V, V)$). Here $!V$ denotes the universal cocommutative counital coalgebra mapping to V, or what is the same, the space of distributions with finite support on (a finite-dimensional subspace of) V viewed as a smooth manifold. This is a bit complicated to describe, but it suffices here to know that there is a canonical (non-linear!) function

$$V \longrightarrow !V\,, \qquad u \longmapsto |\emptyset\rangle_u \tag{2.18}$$

sending a vector $u \in V$ to the Dirac distribution concentrated at u.

Example 2.10. The vector space

$$\begin{aligned} [\![!(\phi_1 \multimap \phi_1) \multimap (\phi_1 \multimap \phi_1)]\!] &= \mathrm{Hom}_{\mathbb{R}}([\![!(\phi_1 \multimap \phi_1)]\!], [\![\phi_1 \multimap \phi_1]\!]) \\ &= \mathrm{Hom}_{\mathbb{R}}(!\,\mathrm{End}_{\mathbb{R}}([\![\phi_1]\!]), \mathrm{End}_{\mathbb{R}}([\![\phi_1]\!])) \end{aligned}$$

is determined once $[\![\phi_1]\!]$ is. The construction can be understood in terms of the tree (**??**) by starting with the vector space $[\![\phi_1]\!]$ at the leaves and combining these vector spaces according to the operations at the vertices.

Having explained how the denotations of types are defined, we now turn to denotations of terms. For any term π of type A the denotation is a vector $[\![\pi]\!] \in [\![A]\!]$. In particular, if ρ is of type $A \multimap B$ then its denotation is a vector

$$[\![\rho]\!] \in [\![A \multimap B]\!] = \mathrm{Hom}_{\mathbb{R}}([\![A]\!], [\![B]\!])$$

that is, the denotation is a linear map $[\![A]\!] \longrightarrow [\![B]\!]$. Similarly, if γ is of type $!A \multimap B$ then its denotation is a linear map

$$[\![\gamma]\!] : ![\![A]\!] \longrightarrow [\![B]\!] .$$

In Remark 2.9 we explained how to think of both ρ, γ as programs taking inputs of type A, but where ρ is restricted to use this input linearly. In the semantics this resolves to the distinction between $[\![A]\!]$ and $![\![A]\!]$. We can still think of $[\![\gamma]\!]$ as taking inputs of type A (i.e. vectors in $[\![A]\!]$) using (2.18) to define the (non-linear) composite

$$[\![A]\!] \longrightarrow ![\![A]\!] \xrightarrow{[\![\gamma]\!]} [\![B]\!] \tag{2.19}$$

which feeds $u \in [\![A]\!]$ to $[\![\gamma]\!]$. This composite is denoted $[\![\gamma]\!]_{nl}$, see [22, Definition 5.10].

We will not describe here the precise rules which construct the denotations of terms, but illustrate the idea with examples from [34, §3]. We fix a type A and set $V = [\![A]\!]$.

Example 2.11. The denotation of a term π of type $\mathbf{int}_A$ is by Example 2.10 a vector

$$[\![\pi]\!] \in [\![\mathbf{int}_A]\!] = \mathrm{Hom}_{\mathbb{R}}(! \, \mathrm{End}_{\mathbb{R}}(V), \mathrm{End}_{\mathbb{R}}(V)) .$$

Thus $[\![\pi]\!]$ is a linear map $! \, \mathrm{End}_{\mathbb{R}}(V) \longrightarrow \mathrm{End}_{\mathbb{R}}(V)$. For each integer $n \geq 0$ there is a term $\underline{n}$ of type $\mathbf{int}_A$, with $\underline{2}$ being given already in (**??**). One calculates that for $\alpha \in \mathrm{End}_{\mathbb{R}}(V)$,

$$[\![\underline{n}]\!] |\emptyset\rangle_{\alpha} = \alpha^n .$$

Another way to express this is to say that the non-linear function

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

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

Example 2.12. The denotation of a proof π of type $\mathbf{bint}_A$ gives a function

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

For every binary sequence $S \in \{0,1\}^*$ there is a corresponding term $\underline{S}$ of type $\mathbf{bint}_A$ [34, §3.2]. 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 $[\![\underline{S}]\!]_{nl}$ sends a pair of linear operators α, β to the product described by S, reading α for 0 and β for 1, and reading in reverse order. For example,

$$[\![\underline{001}]\!]_{nl}(\alpha, \beta) = \beta\alpha\alpha .$$

3 The Linear Logic Recurrent Neural Network

There is a class of augmented RNNs with the following general form: at each time step a linear operator on the hidden state space

$$Z = Z(h^{(t)}, x^{(t+1)}) \in \mathrm{End}_{\mathbb{R}}(\mathscr{H})$$

is generated from the previous hidden state and current input, and then this linear operator Z is applied to the hidden state to generate a vector $Z(h^{(t)})$ which is inserted into the RNN evolution equation for $h^{(t+1)}$. Both the second-order RNN (**??**) and NTM (**??**) have this general form. What is the *generic* augmented RNN of this kind?

Obviously Z should be a (piece-wise) differentiable function of the hidden state and current input, but we consider a more restricted class of functions constructed as follows: in the first step, using feed-forward neural networks, we generate from $h^{(t)}, x^{(t+1)}$ a family of vectors (keeping in mind that tensors and linear operators are also vectors). In the second step, these vectors are combined algorithmically using iteration and the basic operations of linear algebra to produce the linear operator Z. The vectors generated in the first step are called *command vectors*, *data vectors* or *input vectors*, depending on how they are constructed (see below for details). The algorithmic construction in the second step is a fixed linear logic program which we refer to as the *master algorithm*. The generic augmented RNN defined in this way is called the Linear Logic RNN (LLRNN).

In fact it is natural to consider two linear operators $Z_{\mathrm{in}}, Z_{\mathrm{out}}$, with the former appearing within the evolution equation *inside* the nonlinearity and the latter appearing *outside* the nonlinearity. Thus the evolution equation of the LLRNN model is

$$h^{(t+1)} = \sigma\Big(Z_{\mathrm{in}}(h^{(t)}) + Hh^{(t)} + Ux^{(t+1)} + B\Big) + Z_{\mathrm{out}}(h^{(t)}) \tag{3.1}$$

where the linear operators $Z_z \in \mathrm{End}_{\mathbb{R}}(\mathscr{H})$ for $z \in \{\mathrm{in}, \mathrm{out}\}$ depend on $h^{(t)}, x^{(t+1)}$ via the aforementioned feed-forward networks

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

and the denotation of a term $\underline{\mathrm{master}}^z$ in linear logic, via the formula

$$Z_z = [\![\underline{\mathrm{master}}^z]\!]_{nl}\Big(p_1^{(t+1)}, \ldots, p_r^{(t+1)}, b_1^{(t+1)}, \ldots, b_s^{(t+1)}, c^{(t+1)}\Big)\,. \tag{3.2}$$

Here we use that the denotation $[\![\underline{\mathrm{master}}^z]\!]$ defines a differentiable function

$$\mathscr{P}_1 \times \cdots \times \mathscr{P}_r \times \mathscr{B}_1 \times \cdots \times \mathscr{B}_s \times \mathscr{C} \longrightarrow \mathrm{End}_{\mathbb{R}}(\mathscr{H})\,. \tag{3.3}$$

We give the vectors generated in the first step names; we have

- 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.

Definition 3.1. 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

$$!P_1, \ldots, !P_r, !B_1, \ldots, !B_s, !C \vdash A \multimap A\,. \tag{3.4}$$

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

$$\mathscr{P}_i \subseteq [\![P_i]\!]\,, \quad \mathscr{B}_j \subseteq [\![B_j]\!]\,, \quad \mathscr{C} \subseteq [\![C]\!] \tag{3.5}$$

  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*.

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

Proposition 3.2. *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 [34]. □

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 3.3. 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 $!\Gamma, !\Delta, !V \vdash A \multimap A$.

3.1 Examples

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 [22, §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.

Example 3.4 (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

$$Z_{\text{in}} = W^c(x^{(t+1)}) . \tag{3.6}$$

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 3.5 (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 therefore

$$Z_{\text{in}} = \sum_{i=0}^{L} p_i^{(t+1)} \left(W^c(x^{(t+1)})\right)^i . \tag{3.7}$$

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

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

We omit $\underline{\mathrm{master}}^{\mathrm{out}}$ and take $\underline{\mathrm{master}}^{\mathrm{in}}$ to be the proof of

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

given in the term calculus of [42] by

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

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

$$\llbracket \underline{\mathrm{master}}^{\mathrm{in}} \rrbracket_{nl}\big(\sum_i \lambda_i \llbracket \underline{n_i} \rrbracket, \alpha\big) = \sum_i \lambda_i \alpha^{n_i} \,. \tag{3.8}$$

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 (3.7) 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 $\llbracket \underline{\mathrm{master}} \rrbracket_{nl}$ rather than the underlying proof. Nonetheless, the construction of $\underline{\mathrm{master}}$ 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.

3.2 NTM as an LLRNN

We now explain how to present the NTM of Section 2.3 as an LLRNN. We first give the command, data and input types. Let V, W be types of linear logic with $\llbracket V \rrbracket = \mathscr{V}, \llbracket W \rrbracket = \mathscr{W}$, and write $W^\vee = W \multimap 1$ so that $\llbracket W^\vee \rrbracket = \mathscr{W}^*$. The command types Γ and data types Δ are defined to be respectively

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

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

$$\llbracket \underline{m} \rrbracket \in \llbracket \mathbf{int}_W \rrbracket \,, \quad \llbracket \underline{n} \rrbracket \in \llbracket \mathbf{int}_{W^\vee} \rrbracket$$
$$\alpha \in \llbracket W \multimap W \rrbracket = \mathrm{End}_{\mathbb{R}}(\mathscr{W})$$
$$\beta \in \llbracket W^\vee \multimap W^\vee \rrbracket = \mathrm{End}_{\mathbb{R}}(\mathscr{W}^*)$$
$$e \in \llbracket V \multimap V \rrbracket = \mathrm{End}_{\mathbb{R}}(\mathscr{V})$$
$$a \in \llbracket V \rrbracket = \mathscr{V}$$

we have, for $h = (h_0, r, w, M) \in \mathscr{H}$ decomposed according to (2.8),

$$[\![\underline{\text{master}}^{\text{in}}]\!]_{nl}\big([\![\underline{m}]\!], [\![\underline{n}]\!], \alpha, \beta, e, a)(h) = \big(M(r), 0, 0, 0\big)\,,$$
$$[\![\underline{\text{master}}^{\text{out}}]\!]_{nl}\big([\![\underline{m}]\!], [\![\underline{n}]\!], \alpha, \beta, e, a)(h) = \big(0, \alpha^m(r), \beta^n(w), (1_{\mathscr{V}} - e) \circ M + w \otimes a\big)\,.$$

As before we take $\mathscr{V} = \mathscr{H}$. We take

$$\begin{gathered}
\mathscr{P}_1 = \text{span}([\![\underline{0}]\!], \ldots, [\![\underline{N-1}]\!]) \subseteq [\![\mathbf{int}_W]\!]\,,\\
\mathscr{P}_2 = \text{span}([\![\underline{0}]\!], \ldots, [\![\underline{N-1}]\!]) \subseteq [\![\mathbf{int}_{W^\vee}]\!]\,,\\
\mathscr{B}_1 = [\![W \multimap W]\!] = \text{End}_{\mathbb{R}}(\mathscr{W})\,,\\
\mathscr{B}_2 = [\![W^\vee \multimap W^\vee]\!] = \text{End}_{\mathbb{R}}(\mathscr{W}^*)\,,\\
\mathscr{B}_3 = [\![V \multimap V]\!] = \text{End}_{\mathbb{R}}(\mathscr{V})\\
\mathscr{B}_4 = [\![V]\!] = \mathscr{V}\,.
\end{gathered}$$

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 (3.1) 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 (3.1) of the LLRNN reads

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

which agrees with the equations (2.9) – (2.12).

Example 3.6 (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

$$\begin{gathered}
R' : \mathscr{W} \longrightarrow \mathscr{W}\,,\\
R'(\overline{a}) = \overline{-a}.
\end{gathered}$$

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

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

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

$$\begin{aligned}&[\![\underline{\text{master}}^{\text{out}}]\!]_{nl}\big([\![\underline{F}]\!], [\![\underline{G}]\!], \alpha_1, \alpha_2, \beta_1, \beta_2, e, a)(h)\\ &\qquad = \Big(M(r), [\![F]\!]_{nl}(\alpha_1, \alpha_2)(r), [\![G]\!]_{nl}(\beta_1, \beta_2)(w), (1_{\mathscr{V}} - e) \circ M + w \otimes a\Big)\,.\end{aligned}$$

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

4 Generalisations of the NTM

Taking the linear logic point of view on the NTM from Section 3.2 as a starting point, we develop several generalisations. The main purpose of these models is to demonstrate how a high-level idea for extending the basic NTM model can be realised by first translating the idea into linear logic, and then using the LLRNN form of the NTM.

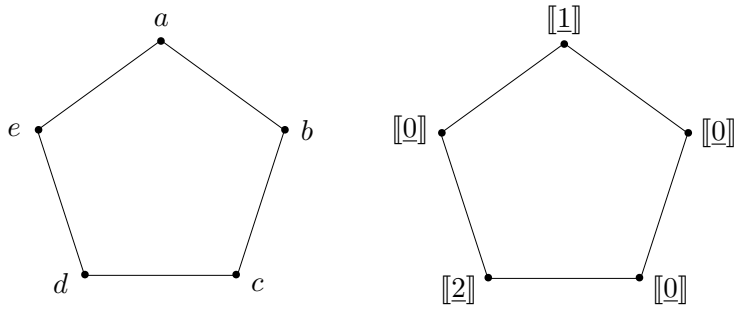
4.1 Pattern NTM

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, which in principle should allow the NTM to learn more complex patterns of memory access. In this example we first present the construction using linear logic, and then afterward translate this into explicit equations involving matrices.

Let $\mathfrak{a} = \{a, b, \ldots, z\}$ and let $\mathscr{V}_1 = \oplus_{i \in \mathfrak{a}} \mathbb{R} i$ be the free vector space on A. We also 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

$$[\![\underline{1}]\!], [\![\underline{0}]\!], [\![\underline{0}]\!], [\![\underline{0}]\!], [\![\underline{2}]\!], [\![\underline{0}]\!], \dots \tag{4.1}$$

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

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

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. The inputs of type $V \multimap V, V$ in Δ are the erase and add vector for the

first memory ring, while the inputs $\mathbf{int}_W \multimap \mathbf{int}_W, \mathbf{int}_W$ give the erase and add vectors for 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

$$g = \big([\![\underline{n}_1]\!], [\![\underline{m}_2]\!], [\![\underline{n}_2]\!], \alpha_1, \beta_1, \alpha_2, \beta_2, e_1, a_1, e_2, a_2\big) \in \bigoplus_{i=1}^{3} \mathscr{P}_i \oplus \bigoplus_{j=1}^{8} \mathscr{B}_j$$

and state vector

$$h = (h_0, r_1, w_1, r_2, w_2, M_1, M_2) \in \mathscr{H} = \mathscr{H}_0 \oplus \mathscr{W} \oplus \mathscr{W}^* \oplus \mathscr{W} \oplus \mathscr{W}^* \oplus \mathscr{S}_1 \oplus \mathscr{S}_2 \tag{4.2}$$

the value of the master functions are

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

where $Q : \mathscr{V}_1 \longrightarrow \mathscr{H}_0$ is a weight matrix, and

$$[\![\underline{\text{master}}^{\text{out}}]\!]_{nl}(g)(h) = \Big(0, [\![\underline{\text{eval}}]\!]\big(M_2(r_2), |\emptyset\rangle_{\alpha_1}\big)(r_1), \beta_1^{n_1}(w_1), \alpha_2^{m_2}(r_2), \beta_2^{n_2}(w_2),$$
$$(1_{\mathscr{V}_1} - e_1) \circ M_1 + \beta_1^{n_1}(w_1) \otimes a_1, (1_{\mathscr{V}_2} - e_2) \circ M_2 + w_2 \otimes a_2\Big)\,.$$

As above we set $\alpha_1 = \alpha_2 = R$ and $\beta_1 = \beta_2 = R^*$. We take $\mathscr{B}_8 = \mathscr{V}_2 \subseteq [\![\mathbf{int}_W]\!]$ and identify $\mathscr{B}_7 \subseteq [\![\mathbf{int}_W \multimap \mathbf{int}_W]\!]$ with $\text{End}_{\mathbb{R}}(\mathscr{V}_2)$ in an appropriate way.

Remark 4.1. The pattern NTM can be presented without linear logic as follows. Suppose we allow rotations given by powers of R in the set $\{0, \ldots, L\}$. Then $\mathscr{V}_2 = \mathbb{R}^{L+1}$ and if we take $N = 5$ as in the above pictures, then $\mathscr{W} = \mathbb{R}^5$. The system is a modified RNN with state space (4.2), and equations (for $1 \leq i \leq 2$ in all cases)

$$s_i^{(t+1)} = \zeta(W_{i,s} h^{(t)} + B_{i,s}) \in \mathscr{W}^* \cong \mathbb{R}^5 \tag{4.3}$$

$$q_2^{(t+1)} = \zeta(W_q h^{(t)} + B_q) \in \mathscr{W}^* \cong \mathbb{R}^5\,, \tag{4.4}$$

$$e_i^{(t+1)} = \sigma(W_{i,e} h^{(t)} + B_{i,e}) \in \text{End}_{\mathbb{R}}(\mathscr{V}_i)\,, \tag{4.5}$$

$$a_i^{(t+1)} = \sigma(W_{i,a} h^{(t)} + B_{i,a}) \in \mathscr{V}_i\,, \tag{4.6}$$

$$w_i^{(t+1)} = \sum_{i=0}^{N-1} s^{(t+1)}(\bar{i}) \cdot (R^*)^i(w_i^{(t)})\,, \tag{4.7}$$

$$r_1^{(t+1)} = \sum_{j=0}^{L} M_2^{(t)}(r_2^{(t)})_j \cdot R^j(r_1^{(t)})\,, \tag{4.8}$$

$$r_2^{(t+1)} = \sum_{i=0}^{N-1} q_2^{(t+1)}(\bar{i}) \cdot R^i(r_2^{(t)})\,, \tag{4.9}$$

$$M_i^{(t+1)} = (1_{\mathscr{V}_i} - e_i^{(t+1)}) \circ M_i^{(t)} + w_i^{(t)} \otimes a_i^{(t+1)}\,. \tag{4.10}$$

Finally, the evolution equation is

$$h_0^{(t+1)} = \sigma\Big(Q\big(M_1^{(t)}(r_1^{(t)})\big) + H_0 h_0^{(t)} + U_0 x^{(t+1)} + B_0\Big)\,. \tag{4.11}$$

The main difference between the ordinary NTM and the pattern NTM is in (4.8) where the content $M_2(r_2) \in \mathbb{R}^{L+1}$ of the second memory ring is used to determine the coefficients of each rotation R^j of the first memory ring's read address.

Remark 4.2. The idea is that the linear logic lets us program algebraic geometry stuff, which pipes linear data around and acts on it. In the context of the pattern NTM the evaluation function gives a morphism of schemes $s : \mathscr{V}_2 \times \mathrm{End}_{\mathbb{R}}(\mathscr{W}) \longrightarrow \mathrm{End}_{\mathbb{R}}(\mathscr{W})$ and evaluation of another kind gives a morphism $\mathscr{S}_2 \times \mathscr{W} \longrightarrow \mathscr{V}_2$.

$$\mathscr{S}_2 \times \mathscr{W}_2 \times G \longrightarrow \mathscr{V}_2 \times G \xrightarrow{1\otimes a} \mathscr{V}_2 \times \mathrm{End}_{\mathbb{R}}(\mathscr{W}) \xrightarrow{\ s\ } \mathrm{End}_{\mathbb{R}}(\mathscr{W})$$

which maps a memory state and read address for the second memory ring, plus a group element (think of it as R) to a linear operator on the read address of the first memory ring.

4.2 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 an example in which, in addition to the original memory ring from the standard NTM, there are two rings with coefficients in $[\![\mathbf{int}_W]\!]$ and a 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. For ease of reference in the sequel we call this the *multiple pattern NTM*.

Before getting into the details, let us give an example of the kind of function this new model is designed to learn. Let $\mathfrak{a} = \{a, b, \ldots, z\} \cup \{\circ, \bullet, S\}$ and $\mathscr{I} = \mathscr{V}_1 = \bigoplus_{i\in\mathfrak{a}} \mathbb{R}i$ where S is the *swap symbol* used to switch between two learned patterns, $\circ$ is the *initial symbol* used by the controller to initialise its memory rings, and $\bullet$ is the *terminal symbol* used to switch the controller to "output" mode. Suppose we are trying to approximate

$$\circ\, abcdeSabcdeSabcde\, \bullet \longmapsto abcde\ aabbccddee\ abcde\,, \tag{4.12}$$

where we insert spaces for clarity in the output. Without being too precise: one pattern is the identity, the other pattern doubles every digit, and at occurrences of S we switch between the patterns. This algorithm is realised using linear logic as follows: if at some time the read address of the second and third rings are focused at positions containing $[\![\underline{i}]\!], [\![\underline{j}]\!]$ respectively and the read address of the fourth ring is focused at $[\![\underline{S}]\!]$ for some

sequence $S \in \{0,1\}^*$ (notation of Example 2.12) then the read address of the first ring will be acted on by R^p where

$$p = ai + bj$$

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

$$\frac{\underline{\text{feed}}}{\vdots}$$
$$\mathbf{int}_W, \mathbf{int}_W, \mathbf{bint}_W \vdash \mathbf{int}_W$$

with linear lambda term

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

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

$$[\![\underline{\text{feed}}]\!](M_2(r_2), M_3(r_3), M_4(r_4))_{nl}(R)(r_1) . \tag{4.13}$$

To be more concrete, suppose $\mathscr{V}_2 = \mathscr{V}_3 = \text{span}([\![\underline{0}]\!], \ldots, [\![\underline{L}]\!])$ and

$$\mathscr{V}_4 = \text{span}([\![\underline{S}_1]\!], \ldots, [\![\underline{S}_m]\!])$$

where $S_k \in \{0,1\}^*$ for $1 \le k \le m$. We write a_k, b_k for, respectively, the number of 0's and 1's in S_k. At any given time the controller observes some linear combination of the contents of the rings

$$M_2(r_2) = \sum_{i=0}^{L} \lambda_i [\![\underline{i}]\!] , \qquad M_3(r_3) = \sum_{j=0}^{L} \mu_j [\![\underline{j}]\!] , \qquad M_4(r_4) = \sum_{k=0}^{m} \kappa_k [\![\underline{S}_k]\!] . \tag{4.14}$$

In this case the term in (4.13) is

$$\sum_{i,j,k} \lambda_i \mu_j \kappa_k R^{a_k i + b_k j}(r_1) \tag{4.15}$$

so that the time evolution of the first read address will become (cf. (2.13))

$$r_1[u] \longleftarrow \sum_{i,j,k} \lambda_i \mu_j \kappa_k r_1[u - a_k i - b_k j] . \tag{4.16}$$

Returning to the example, set $\mathscr{W}_i = (\mathbb{Z}/N_i\mathbb{Z})^{\mathbb{R}}$ with $N_1 > 18$ and $N_2 = N_3 = N_4 = 2$. The particular values are not important, the memory rings need only be sufficiently large. One method (described anthropomorphically) for the controller to approximate (4.12) in the above system is to perform the following steps:

1. Conditioned on the initial symbol $\circ$ (or a sequence of distinct initial symbols) the controller inserts $[\![\underline{1}]\!]$ at every position of the second memory ring and the repeating pattern $[\![\underline{0}]\!], [\![\underline{1}]\!]$ on both the third and fourth rings (note that in the first case these are of type $\mathbf{int}_W$ and in the second case of type $\mathbf{bint}_W$).

2. The controller writes the input sequence $a, b, c, d, e, S, \ldots, \bullet$ including the terminal symbol but not the initial symbol, onto the first memory ring.

3. The controller switches into output mode, with the read addresses of all memory rings sharply focused at the zero position. In particular, the fourth memory ring is focused at $[\![\underline{0}]\!]$ and the pattern on the second memory ring (the identity) is "active".

4. The controller advances the read address of the second memory ring one position in each time step, yielding $[\![\underline{1}]\!], [\![\underline{1}]\!], \ldots$ and so advances the read address of the first memory ring one position per unit time, emitting $V(a), V(b), V(c), V(d), V(e)$.

5. The first encounter with the vector $V(S)$ triggers the controller to

   (i) advance the read address of the fourth memory ring so that the second pattern becomes "active", and

   (ii) begin advancing the read address of the third ring rather than the second.

6. The controller advances the read address of the third memory ring one position in each time step, yielding $[\![\underline{0}]\!], [\![\underline{1}]\!], [\![\underline{0}]\!], \ldots$ and so advances the read address of the first memory ring by zero positions (i.e. stays stationary), then by one position, then by zero, etc., emitting $V(a), V(a), V(b), V(b), \ldots$.

7. Finally, when the controller emits the vector $V(\bullet)$ we stop.

4.3 Polynomial step NTM

For our final example extends the pattern NTM of Section 4.1 with a modal symbol T which modifies the way the pattern is interpreted. The alphabet is therefore

$$\mathfrak{a} = \{a, b, \ldots, z\} \cup \{\circ, \bullet, T\} \tag{4.17}$$

with $\mathscr{I} = \mathscr{V}_1 = \bigoplus_{i \in \mathfrak{a}} \mathbb{R}i$ as before. The behaviour of the modal symbol T can be modelled in the LLRNN framework by adding a memory ring with coefficients in $[\![!\mathbf{int}_W \multimap \mathbf{int}_W]\!]$.

Any polynomial $a(x) \in \mathbb{N}[x]$ can be encoded by a term $\underline{a}$ of $!\mathbf{int}_W \multimap \mathbf{int}_W$. We therefore have three memory rings with coefficient spaces

$$\mathscr{V}_1 = \bigoplus_{i \in \mathfrak{a}} \mathbb{R} i \,, \quad \mathscr{V}_2 \subseteq [\![\mathbf{int}_W]\!] \,, \quad \mathscr{V}_3 \subseteq [\![!\mathbf{int}_W \multimap \mathbf{int}_W]\!] \,.$$

Take $\mathscr{V}_2$ as in Section 4.1, and

$$\mathscr{V}_3 = \mathrm{span}([\![\underline{a_1}]\!], \ldots, [\![\underline{a_k}]\!])$$

for some polynomials $a_1, \ldots, a_k \in \mathbb{N}[x]$. The relevant part of the master algorithm is

$$\frac{\underline{\text{newfeed}}}{\vdots}$$
$$\mathbf{int}_W, !\mathbf{int}_W \multimap \mathbf{int}_W \vdash \mathbf{int}_W$$

with linear lambda term

$$\underline{\text{newfeed}} = (\lambda pqx.((q(\text{promote } p))x)) \,. \tag{4.18}$$

This appears in $[\![\underline{\text{master}}^{\text{out}}]\!]_{nl}(g)(h)$ in the form

$$[\![\underline{\text{newfeed}}]\!](M_2(r_2), M_3(r_3))_{nl}(R)(r_1) \,. \tag{4.19}$$

To give an example: if $M_2(r_2) = \sum_i \lambda_i [\![\underline{i}]\!]$ and $M_3(r_3) = \sum_j \mu_j [\![\underline{a_j}]\!]$, then (4.19) is

$$\Big\{ \sum_j \mu_j a_j \big(\sum_i \lambda_i R^i \big) \Big\}(r_1) \,. \tag{4.20}$$

Note that $\mathrm{End}_{\mathbb{R}}(\mathscr{W})$ is an algebra, so $a_j(Y)$ is defined for any linear operator Y. If the third memory ring is sharply focused at the identity polynomial $1 \in \mathbb{N}[x]$ then (4.20) is just $\sum_i \lambda_i R^i(r_1)$, so we shift the first memory ring according to the pattern on the second memory ring. If however the third memory ring is sharply focused at $a(x) = x^2$ then for every step of size $s \in \mathbb{N}$ dictated by the pattern on the second memory ring, we actually execute a step of size s^2.

4.3.1 Dihedral polynomial step NTM

There is a variant of the polynomial step NTM where we use binary integers instead of integer types, following Example 3.6. We keep the same alphabet, but take

$$\mathscr{V}_1 = \bigoplus_{i \in \mathfrak{a}} \mathbb{R} i \,, \quad \mathscr{V}_2 \subseteq [\![\mathbf{bint}_W]\!] \,, \quad \mathscr{V}_3 \subseteq [\![!\mathbf{bint}_W \multimap \mathbf{bint}_W]\!] \,.$$

Take $\mathscr{V}_2$ as in Section 4.1, and

$$\mathscr{V}_3 = \mathrm{span}([\![\underline{\mathrm{id}}]\!], [\![\underline{\text{repeat}}]\!]) \,.$$

Where id is the identity function, and repeat repeats a binary integer [34, §3.2]. Recall that a binary integer, say 001, is interpreted as a word $R'RR$ in the dihedral group and acts on the first memory ring according to the action of this group on $\mathscr{W}$. A sequence of binary integers stored on the second memory ring may therefore be thought of as a sequence of group elements $g_0, \ldots, g_4$. If the third memory ring is sharply focused at the identity function, this sequence will be applied as it stands. If, however, the third memory ring is sharply focused at the repeat function, the sequence that gets applied will be $g_0^2, \ldots, g_4^2$ as the underlying binary sequence being repeated

$$[\![\underline{\text{repeat}}]\!]_{nl}[\![\underline{001}]\!] = [\![\underline{001001}]\!]$$

corresponds to the group elements being squared

$$R'RR \longmapsto R'RRR'RR\,.$$

5 Unpolished remarks

Remark 5.1. A more powerful system would substitute a neural theorem prover along the lines of [30, 32, 44] 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 [31].

Remark 5.2. In functional programming languages like differential λ-calculus [38] and differential linear logic [33] 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 [37] which is widely used to compute derivatives of complicated real-valued functions, for example in the backpropagation algorithms of machine learning packages like TensorFlow [35, §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 [39]. 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.

Remark 5.3. One method of adding algorithmic elements to deep neural networks that has been explored extensively is the idea of *attention networks* which implement a form of "distributed decision making" [2, p.1].

According to [2, §III.A] one of the problems with the original encoder-decoder network approach to machine translation [17] is that the model compresses the information in the input sequence into a vector without any presupposed structure; intuitively this makes

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

the representations learned by the model harder to understand, and the output sequence harder to construct. From this perspective the LLRNN versions of the NTM models in previous sections can be thought of as structured encoder-decoder models with attention, where the representation learned by the encoder is the vector state of the memory rings after all input symbols have been read, as well as the matrix V mapping symbols to hidden state vectors. In our examples these vectors are highly structured, in the precise sense that the coefficients are highly correlated under the dynamics of the RNN.

These correlations ultimately arise from the fact that the coefficient spaces of the memory rings are spanned by linear logic denotations which imbue the coefficients with a dynamic meaning. Thus linear logic provides a structured algorithmic "scaffolding" which is populated by the encoder. For example, in Section 4.1 the meaning of the symbol S (which is implicit in the training data) is made explicit by learning the contents of the fourth memory ring, and the association between S and a manipulation of this ring, which is encoded by the coefficients of $V(S)$.

Remark 5.4. What is the link between attention mechanisms and the vanishing gradient problem? Clearly one of the purposes of attention mechanisms and external memories is to avoid the problem of "forgetting" large scale structure in the input to a recurrent network. The explicit way to do this is to restrict one part of the recurrent matrix H to be the identity [45]. More generally, we could restrict a part of the current matrix to be an element of the orthogonal group, i.e. a rotation. But then if look at that matrix, one part of it is precisely predicting a "rotation" to apply to the other part, and we are basically in the situation of a NTM.

We write $\Phi = [\![\text{less}]\!]$ for the proof of Definition 5.7.

Lemma 5.5. *There is a proof* $\underline{\text{more}}$ *such that denotation gives a linear map*

$$\Psi = [\![\text{more}]\!] : [\![\boldsymbol{bint}_W]\!] \longrightarrow [\![\boldsymbol{bint}_{W\&W}]\!]$$

with the following properties:

(i) $\Phi \circ \Psi = 1$.

(ii) On the subspace of block matrices

$$B = \left\{ \begin{pmatrix} \gamma_1 & 0 \\ 0 & \gamma_2 \end{pmatrix} \;\middle|\; \gamma_i \in \mathrm{End}_{\mathbb{R}}(\mathscr{W}) \right\} \subseteq \mathrm{End}_{\mathbb{R}}(\mathscr{W} \oplus \mathscr{W})$$

we have

$$\Psi\big([\![\underline{S}_W]\!]\big)|_{B\times B} = [\![\underline{S}_{W\&W}]\!]|_{B\times B} \tag{5.1}$$

for all $S \in \{0,1\}^*$.

Proof. Let π_i, ι_i denote the projections from and injections into $[\![W\&W]\!] = \mathscr{W} \oplus \mathscr{W}$. We define for $p \in [\![\mathbf{bint}_W]\!]$

$$\Psi(p)(\alpha, \beta) = \big(p(\pi_1\alpha\iota_1, \pi_1\beta\iota_1), p(\pi_2\alpha\iota_2, \pi_2\beta\iota_2)\big)\,.$$

It is clear that this the denotations of a proof. We observe that

$$\begin{aligned}\big(\Phi\Psi(p)\big)(\alpha,\beta) &= \pi_1\Psi(p)(\alpha \oplus 1_{\mathscr{W}}, \beta \oplus 1_{\mathscr{W}}) \\ &= \pi_1\Big(p(\pi_1(\alpha \oplus 1_{\mathscr{W}})\iota_1, \pi_1(\beta \oplus 1_{\mathscr{W}})\iota_1), p(\pi_2(\alpha \oplus 1_{\mathscr{W}})\iota_2, \pi_2(\beta \oplus 1_{\mathscr{W}})\iota_2\Big) \\ &= \pi_1\big(p(\alpha,\beta), p(1_{\mathscr{W}}, 1_{\mathscr{W}})\big) \\ &= p(\alpha,\beta)\end{aligned}$$

which proves (i). It is clear that (ii) holds. □

Remark 5.6. The purpose of linear logic in this context is to give a coherent and general way of writing down such polynomials, which realise a given computational idea when inserted into the evolution equation of an RNN.

In this case the term in (4.13) is

$$\big(\sum_i \lambda_i R^i\big)^a \big(\sum_j \mu_j R^j\big)^b (r_1) \tag{5.2}$$

which can be expanded as

$$\sum_{u_0+\cdots+u_L=a} \sum_{v_0+\cdots+v_L=b} \frac{a!}{u_0!\cdots u_L!}\frac{b!}{v_0!\cdots v_L!}\lambda_0^{u_0}\cdots\lambda_L^{u_L}\mu_0^{v_0}\cdots\mu_L^{v_L}R^{\sum_t t(u_t+v_t)}(r_1)\,. \tag{5.3}$$

This may be written as $\sum_p f_p(\lambda,\mu)R^p(r_1)$ for some polynomials f_p in variables $\{\lambda_i, \mu_j\}_{i,j}$.

5.1 Decaying pattern NTM

We take the same alphabet $\mathfrak{a}$ as in the multiple pattern NTM, and the same coefficient space $\mathscr{V}_1$ for the first memory ring. There are two additional memory rings, with

$$\mathscr{V}_2 \subseteq [\![\mathbf{bint}_{W\&W}]\!]\,, \qquad \mathscr{V}_3 \subseteq [\![\mathbf{bint}_{W\&W} \multimap \mathbf{bint}_W \,\&\, \mathbf{bint}_W]\!]\,.$$

An example of a proof in $\mathscr{V}_3$ is the algorithm $\underline{\text{tail}}$ which takes a binary sequence and returns a pair by separating the tail, for example

$$[\![\underline{\text{tail}}]\!]\big(\underline{1101}_{W\&W}\big) = (\underline{110}_W, \underline{1}_W)\,.$$

The idea is that if the third memory ring is sharply focused at $\underline{\text{tail}}$ then we update the second memory ring by replacing a vector $[\![\underline{1101}]\!]$ at a vertex of the ring by its truncation $[\![\underline{110}]\!]$, and we moreover use the tail $[\![\underline{1}]\!]$ to update the first memory ring by reading it as a power of the rotation operator. In this way the contents of the second memory ring *decay* over time. To realise this we need the following auxiliary maps:

Definition 5.7. Given a type W with $\mathscr{W} = [\![W]\!]$ let $\underline{\text{less}}$ be the proof of

$$\mathbf{bint}_{W\&W} \vdash \mathbf{bint}_W$$

with the property that for $\alpha, \beta \in \mathrm{End}_{\mathbb{R}}([\![W]\!])$ and $q \in [\![\mathbf{bint}_{W\&W}]\!]$

$$[\![\underline{\text{less}}]\!](q)(\alpha, \beta) = \pi_1 q(\alpha \oplus 1_{\mathscr{W}}, \beta \oplus 1_{\mathscr{W}})$$

where π_i denote the projections from $\mathscr{W} \oplus \mathscr{W}$. Then

$$\Phi[\![\underline{S}_{W\&W}]\!] = [\![\underline{S}_W]\!]$$

for all $S \in \{0,1\}^*$.

There does not appear to be a proof $\underline{\text{more}}$ in linear logic which gives a section of $\underline{\text{less}}$ in the sense that its denotation maps $[\![\underline{S}_W]\!]$ to $[\![\underline{S}_{W\&W}]\!]$ (see however Lemma 5.5). But we can just fix one ad hoc (this is simulating second order linear logic in an ugly way).

Definition 5.8. We fix a linear map $\psi : [\![\mathbf{bint}_W]\!] \longrightarrow [\![\mathbf{bint}_{W\&W}]\!]$ with the property that

$$\psi[\![\underline{S}_W]\!] = [\![\underline{S}_{W\&W}]\!]$$

for a finite collection of sequences S that are relevant to the model.

The master algorithm at a given time step applies $M_3(r_3) \in \mathscr{V}_3$ to $M_2(r_2) \in \mathscr{V}_2$ to get

$$[\![\underline{\text{feed}}]\!](M_2(r_2), M_3(r_3)) \in [\![\mathbf{bint}_W]\!] \oplus [\![\mathbf{bint}_W]\!]\,.$$

The second component is then applied to the contents of the first memory ring, as in

$$r_1^{(t+1)} = \Big(\pi_2[\![\underline{\text{feed}}]\!](M_2(r_2), M_3(r_3))\Big)(1_{\mathscr{W}}, R)(r_1^{(t)})\,.$$

Simultaneously, we update the memory state of the second ring according to the following rule: it becomes the linear map $\mathscr{W} \longrightarrow \mathscr{V}_2$ defined by

$$w \longmapsto \psi\pi_1[\![\underline{\text{feed}}]\!](M_2(w), M_3(r_3))\,.$$

This kind of structure could in principle be used by the controller to predict that an event will happen a fixed amount of time in the future, by pushing "countdowns" $[\![\underline{10000}]\!]$ onto the memory ring, and have the controller react to a symbol coming off the first memory ring after it is shifted by the final $[\![\underline{1}]\!]$.

Remark 5.9. Actually we seem to have been confused about the kinds of polynomials we get. It seems that unless $\underline{\text{feed}}$ uses non-linear inputs we never get powers of variables. So we should find an example where this is the case.

References

[1] D. Bahdanau, K. Cho and Y. Bengio, *Neural machine translation by jointly learning to align and translate*, arXiv preprint arXiv:1409.0473.

[2] K. Cho, A. Courville and Y. Bengio, *Describing multimedia content using attention-based encoder-decoder networks*, IEEE Transactions on Multimedia, 17(11), 1875-1886, (2015).

[3] A. Neelakantan, Q. V. Le and I. Sutskever, *Neural Programmer: Inducing Latent Programs with Gradient Descent*, (2015).

[4] S. Reed and N.de Freitas, *Neural Programmer-Interpreters*, (2016).

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

[6] 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

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

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

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

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

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

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

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

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

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

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

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

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

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

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

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

[29] 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.

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

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

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

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

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

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

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

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

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

[39] 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].

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

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

[42] 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.

[43] I. Goodfellow, Y. Bengio and A. Courville, *Deep learning*, MIT Press.

[44] Loos, S., Irving, G., Szegedy, C., Kaliszyk, C. (2017). Deep Network Guided Proof Search. arXiv preprint arXiv:1701.06972.

[45] T. Mikolov, A. Joulin, S. Chopra, M. Mathieu and M. A. Ranzato, *Learning longer memory in recurrent neural networks*, arXiv preprint arXiv:1412.7753 (2014).