

Linear logic and recurrent neural networks

Huiyi Hu, Daniel Murfet

February 8, 2017

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] and multiplicative RNN [16].

2 Architecture

2.1 The second-order RNN

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

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

and by ζ the *softmax* function

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

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

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

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

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

Suppose the input vector space is \mathcal{I} and the hidden state space is \mathcal{H} , so that $x^{(t)} \in \mathcal{I}$ and $h^{(t)} \in \mathcal{H}$ for all t . The value of the RNN on a sequence $\mathbf{x} = (x^{(1)}, \dots, x^{(T)}) \in \mathcal{I}^{\oplus T}$ is computed by applying (2.1) for $0 \leq t \leq T-1$, accumulating the values $\mathbf{y} = (y^{(1)}, \dots, y^{(T)})$ 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 [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 i th coordinate is given by the formula

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

where V is interpreted as a tensor in

$$(2.4) \quad \mathcal{I}^* \otimes \mathcal{H}^* \otimes \mathcal{H} \cong \text{Hom}_{\mathbb{R}}(\mathcal{I}, \text{End}_{\mathbb{R}}(\mathcal{H})).$$

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

Definition 2.2. The second-order RNN [20, 21] is defined by weights H, U, B, V and

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

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 $\dim(\text{End}_{\mathbb{R}}(\mathcal{H})) = \dim(\mathcal{H})^2$. The *multiplicative RNN* is introduced in [16] as a more tractable model. The central idea is the same, but an auxiliary space \mathcal{K} is introduced and the RNN learns three linear maps

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

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

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

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 2.3. 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)

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

The relevance of this “compositionality” to NLP is remarked on in [15].

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 the hidden state; this operator moreover depends *linearly* on the input. The LLRNN model is similar in 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 [22, 34] 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 $\llbracket - \rrbracket$ of linear logic in the category \mathcal{V} of vector spaces from [22, §5.1, §5.3] is defined as follows: for a variable x the denotation $\llbracket x \rrbracket$ is a chosen finite-dimensional vector space, and

$$\begin{aligned}\llbracket A \multimap B \rrbracket &= \text{Hom}_k(\llbracket A \rrbracket, \llbracket B \rrbracket), \\ \llbracket A \otimes B \rrbracket &= \llbracket A \rrbracket \otimes \llbracket B \rrbracket, \\ \llbracket !A \rrbracket &= !\llbracket A \rrbracket,\end{aligned}$$

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

For any proof π of type A the denotation is a vector $\llbracket \pi \rrbracket \in \llbracket A \rrbracket$. Moreover to a proof π of type $!A \multimap B$ we naturally associate [22, Definition 5.10] a function $\llbracket \pi \rrbracket_{nl} : \llbracket A \rrbracket \longrightarrow \llbracket B \rrbracket$.

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

- types P_1, \dots, P_r called the *command types*;
- types B_1, \dots, B_s , called the *data types*;
- a type C , called the *input type*;

- a type A with $\llbracket A \rrbracket = \mathcal{H}$, the hidden state space;
- two proofs $\underline{\text{master}}^z$ for $z \in \{\text{in}, \text{out}\}$ of the sequent

$$(2.8) \quad !P_1, \dots, !P_r, !B_1, \dots, !B_s, !C \vdash A \multimap A.$$

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

$$(2.9) \quad \mathcal{P}_i \subseteq \llbracket P_i \rrbracket, \quad \mathcal{B}_j \subseteq \llbracket B_j \rrbracket, \quad \mathcal{C} \subseteq \llbracket C \rrbracket$$

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

From this data we obtain the following functions by restriction:

$$(2.10) \quad \llbracket \underline{\text{master}}^z \rrbracket_{nl} : \mathcal{P}_1 \times \dots \times \mathcal{P}_r \times \mathcal{B}_1 \times \dots \times \mathcal{B}_s \times \mathcal{C} \longrightarrow \text{End}_{\mathbb{R}}(\mathcal{H}).$$

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

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

and the evolution equation

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

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

$$(2.12) \quad Z_z = \llbracket \underline{\text{master}}^z \rrbracket_{nl}\left(p_1^{(t+1)}, \dots, p_r^{(t+1)}, b_1^{(t+1)}, \dots, b_s^{(t+1)}, c^{(t+1)}\right)(h^{(t)}).$$

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

Proposition 2.5. *The functions $\llbracket \underline{\text{master}}^z \rrbracket_{nl}$ are smooth.*

Proof. This follows from the hypothesis that $\mathcal{P}_i, \mathcal{B}_j$ are generated by denotations of linear logic proofs, and the smoothness of these denotations [34]. \square

Moreover, the derivatives of $\llbracket \underline{\text{master}}^z \rrbracket_{nl}$ 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 $\Gamma = P_1, \dots, P_r$ for the command types and $\Delta = B_1, \dots, B_s$ 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, $!\Gamma = !P_1, \dots, !P_r$. 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 [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.

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 $\alpha \in \text{End}_{\mathbb{R}}(\mathcal{H})$ 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 $\llbracket A \rrbracket = \mathcal{H}$. 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

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

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 [8]. There are two ways of presenting proofs in linear logic: proof trees, and the term calculus of [42] 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 [34, §3].

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

$$\llbracket \pi \rrbracket_{nl} : \text{End}_{\mathbb{R}}(\llbracket A \rrbracket) \longrightarrow \text{End}_{\mathbb{R}}(\llbracket A \rrbracket)$$

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 [42]

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

The denotation satisfies $\llbracket \underline{n} \rrbracket_{nl}(X) = X^n$.

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

$$\llbracket \pi \rrbracket_{nl} : \text{End}_{\mathbb{R}}(\llbracket A \rrbracket) \times \text{End}_{\mathbb{R}}(\llbracket A \rrbracket) \longrightarrow \text{End}_{\mathbb{R}}(\llbracket A \rrbracket).$$

For every binary sequence $S \in \{0, 1\}^*$ there is a corresponding proof \underline{S} of \mathbf{bint}_A [34, §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 $\llbracket \underline{S} \rrbracket$ 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,

$$\llbracket \underline{001} \rrbracket(X, Y) = YXX .$$

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

$$\mathcal{C} = \text{End}_{\mathbb{R}}(\mathcal{H}) = \llbracket A \multimap A \rrbracket .$$

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 \mathcal{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 $\llbracket \underline{\text{master}}^{\text{in}} \rrbracket_{nl}(\alpha) = \alpha$ and hence

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

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 $\{0, \dots, L\}$. Then at each time step the RNN will generate a distribution $p^{(t+1)}$ over $\{0, \dots, L\}$ 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

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

The operation of taking a linear operator and raising it to the n th 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

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

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

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

given in the term calculus of [42] by

$$\underline{\text{master}}^{\text{in}} = (\lambda n. (\lambda a. (\text{derelect}(n) a))) .$$

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

$$(3.3) \quad \llbracket \underline{\text{master}}^{\text{in}} \rrbracket_{nl} \left(\sum_i \lambda_i \llbracket n_i \rrbracket, \alpha \right) = \sum_i \lambda_i \alpha^{n_i} .$$

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. 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 Neural Turing Machine (NTM) [11] is an RNN augmented with a second hidden state space that functions like an external memory. This memory is read from and written to using attention vectors which are manipulated based on the hidden state of the RNN. To describe the precise form of this manipulation, it is important to know that the memory state can be viewed as a sequence of vectors arranged at the vertices of a regular polygon. An *attention vector* is by definition a probability distribution over the vertices of this polygon. There is an natural action of rotation R on both memory states and attention vectors, and the read and write attention vectors are manipulated by predicting, at each time step, which power of the linear operator R to apply. From this point of view, the NTM with location-based addressing may be described naturally using the LLRNN.

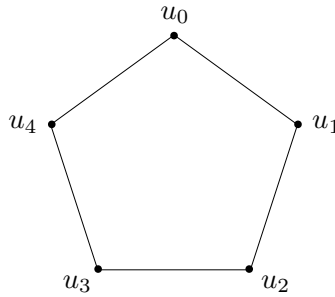
To explain, we first introduce the *address space*

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

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

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

A memory state M is a linear map $M : \mathcal{W} \longrightarrow \mathcal{V}$ 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 [11] we may take $\mathcal{W} = \mathbb{R}^N$ and $\mathcal{V} = \mathbb{R}^M$ so that elements of \mathcal{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 \mathcal{W} and therefore on \mathcal{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 $R : \mathcal{W} \longrightarrow \mathcal{W}$ be the linear map defined by

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

with dual map $R^* : \mathcal{W}^* \longrightarrow \mathcal{W}^*$. 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

$$(3.4) \quad \mathcal{H} = \mathcal{H}_0 \oplus \mathcal{W} \oplus \mathcal{W}^* \oplus \mathcal{S}$$

and the state at time t is written

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

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

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

In practice we will assume that $e^{(t+1)}$ is diagonal with respect to a chosen basis of \mathcal{V} . The update equation for the addresses [11, Eq. (8)] and memory [11, Eq. (3),(4)] are

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

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

$$(3.7) \quad M^{(t+1)} = (1_{\mathcal{V}} - e^{(t+1)}) \circ M^{(t)} + w^{(t)} \otimes a^{(t+1)}.$$

Finally, the evolution equation is

$$(3.8) \quad h_0^{(t+1)} = \sigma\left(V(M^{(t)}(r^{(t)})) + H_0 h_0^{(t)} + U_0 x^{(t+1)} + B_0\right).$$

where

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

and $B_0 \in \mathcal{H}_0$ are weight matrices. Following [11, §3.2] we write e for the *erase vector* and a for the *add vector*. 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

¹We omit the sharpening step [11, Eq. (9)] for simplicity. The original NTM paper [11] 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 [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).

3.1.1 NTM as an LLRNN

We now explain how to present the restricted NTM of the previous section as a LLRNN. We first give the command, data and input types. Let V, W be types of linear logic with $\llbracket V \rrbracket = \mathcal{V}$, $\llbracket W \rrbracket = \mathcal{W}$, and write $W^\vee = W \multimap 1$ so that $\llbracket W^\vee \rrbracket = \mathcal{W}^*$.

The command types Γ and data types Δ are defined to be respectively

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

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

$$\begin{aligned} \llbracket \underline{m} \rrbracket &\in \llbracket \mathbf{int}_W \rrbracket, & \llbracket \underline{n} \rrbracket &\in \llbracket \mathbf{int}_{W^\vee} \rrbracket \\ \alpha &\in \llbracket W \multimap W \rrbracket = \text{End}_{\mathbb{R}}(\mathcal{W}) \\ \beta &\in \llbracket W^\vee \multimap W^\vee \rrbracket = \text{End}_{\mathbb{R}}(\mathcal{W}^*) \\ e &\in \llbracket V \multimap V \rrbracket = \text{End}_{\mathbb{R}}(\mathcal{V}) \\ a &\in \llbracket V \rrbracket = \mathcal{V} \end{aligned}$$

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

$$\begin{aligned} \llbracket \text{master}^{\text{in}} \rrbracket_{nl}(\llbracket \underline{m} \rrbracket, \llbracket \underline{n} \rrbracket, \alpha, \beta, e, a)(h) &= (V(M(r)), 0, 0, 0), \\ \llbracket \text{master}^{\text{out}} \rrbracket_{nl}(\llbracket \underline{m} \rrbracket, \llbracket \underline{n} \rrbracket, \alpha, \beta, e, a)(h) &= (0, \alpha^m(r), \beta^n(w), (1_{\mathcal{V}} - e) \circ M + w \otimes a). \end{aligned}$$

For the lambda term, see Appendix ?? . We take

$$\begin{aligned} \mathcal{P}_1 &= \text{span}(\llbracket 0 \rrbracket, \dots, \llbracket N-1 \rrbracket) \subseteq \llbracket \mathbf{int}_W \rrbracket, \\ \mathcal{P}_2 &= \text{span}(\llbracket 0 \rrbracket, \dots, \llbracket N-1 \rrbracket) \subseteq \llbracket \mathbf{int}_{W^\vee} \rrbracket, \\ \mathcal{B}_1 &= \llbracket W \multimap W \rrbracket = \text{End}_{\mathbb{R}}(\mathcal{W}), \\ \mathcal{B}_2 &= \llbracket W^\vee \multimap W^\vee \rrbracket = \text{End}_{\mathbb{R}}(\mathcal{W}^*), \\ \mathcal{B}_3 &= \llbracket V \multimap V \rrbracket = \text{End}_{\mathbb{R}}(\mathcal{V}) \\ \mathcal{B}_4 &= \llbracket V \rrbracket = \mathcal{V}. \end{aligned}$$

We restrict the functions $\llbracket \text{master}^z \rrbracket_{nl}$ 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 $\{\llbracket i \rrbracket\}_{i=0}^{N-1}$ of $\mathcal{P}_1, \mathcal{P}_2$ 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 \mathcal{H} to \mathcal{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

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

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

$$\begin{aligned} T : \mathcal{W} &\longrightarrow \mathcal{W}, \\ T(\bar{a}) &= \overline{-a}. \end{aligned}$$

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

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

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

$$\begin{aligned} &\llbracket \underline{\text{master}}^{\text{out}} \rrbracket_{nl}(\llbracket F \rrbracket, \llbracket G \rrbracket, \alpha_1, \alpha_2, \beta_1, \beta_2, e, a)(h) \\ &= \left(V(M(r)), \llbracket F \rrbracket_{nl}(\alpha_1, \alpha_2)(r), \llbracket G \rrbracket_{nl}(\beta_1, \beta_2)(w), (1_{\mathcal{V}} - e) \circ M + w \otimes a \right). \end{aligned}$$

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

The Differentiable Neural Computer (DNC) [5] 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.

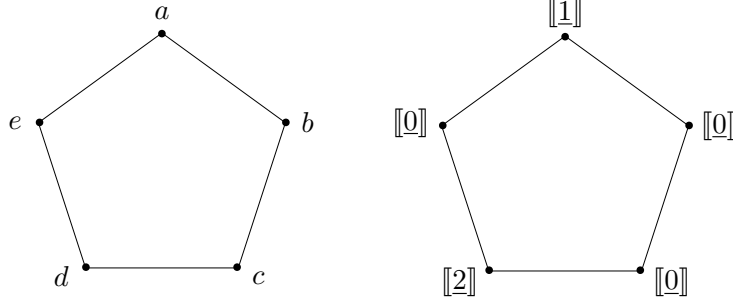
3.2 Pattern NTM

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 $\llbracket \underline{n} \rrbracket$ representing a clockwise angle of $\frac{2\pi n}{5}$) the pattern may be stored as a sequence of vectors in $\llbracket \mathbf{int}_W \rrbracket$. 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 $\mathbf{a} = \{a, b, \dots, z\}$ and let $\mathcal{V}_1 = \oplus_{i \in \mathbf{a}} \mathbb{R}i$ be the free vector space on A . We also take a finite-dimensional subspace $\mathcal{V}_2 \subseteq \llbracket \mathbf{int}_W \rrbracket$. For convenience we keep the same address space for both memory rings. So we have memory spaces for $i \in \{1, 2\}$

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

An example of a combined state $\mathcal{S} = \mathcal{S}_1 \oplus \mathcal{S}_2$ 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

$$(3.9) \quad \llbracket 1 \rrbracket, \llbracket 0 \rrbracket, \llbracket 0 \rrbracket, \llbracket 0 \rrbracket, \llbracket 2 \rrbracket, \llbracket 0 \rrbracket, \dots$$

will be used to control the read address of the first memory ring. More precisely, the operator applied to the read address $r_1 \in \mathcal{W}$ 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 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

$$\llbracket \underline{\text{eval}} \rrbracket \left(\sum_i \lambda_i \llbracket n_i \rrbracket, |\emptyset\rangle_\alpha \right) = \sum_i \lambda_i \alpha^{n_i}.$$

The first argument to this evaluation function will be the state $M_2(r_2) = \sum_i \lambda_i \llbracket n_i \rrbracket$ 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

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

The command and data types are

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

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. 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

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

and state vector

$$(3.10) \quad h = (h_0, r_1, w_1, r_2, w_2, M_1, M_2) \in \mathcal{H} = \mathcal{H}_0 \oplus \mathcal{W} \oplus \mathcal{W}^* \oplus \mathcal{W} \oplus \mathcal{W}^* \oplus \mathcal{S}_1 \oplus \mathcal{S}_2$$

the value of the master functions are

$$\llbracket \underline{\text{master}}^{\text{in}} \rrbracket_{nl}(g)(h) = (V(M_1(r_1)), 0, 0, 0, 0, 0, 0).$$

and

$$\begin{aligned}\llbracket \underline{\text{master}}^{\text{out}} \rrbracket_{nl}(g)(h) &= \left(0, \llbracket \underline{\text{eval}} \rrbracket(M_2(r_2), |\emptyset\rangle_{\alpha_1})(r_1), \beta_1^{n_1}(w_1), \alpha_2^{m_2}(r_2), \beta_2^{n_2}(w_2), \right. \\ &\quad \left. (1_{\gamma_1} - e_1) \circ M_1 + \beta_1^{n_1}(w_1) \otimes a_1, (1_{\gamma_2} - e_2) \circ M_2 + w_2 \otimes a_2 \right).\end{aligned}$$

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

3.2.1 An alternative presentation

The pattern NTM can be presented without mentioning linear logic as follows. Suppose we allow rotations given by powers of R in the set $\{0, \dots, L\}$. Then $\mathcal{V}_2 = \mathbb{R}^{L+1}$ and if we take $N = 5$ as in the above pictures, then $\mathcal{W} = \mathbb{R}^5$. The system is a modified RNN with

state space (3.10), and equations (for $1 \leq i \leq 2$ in all cases)

$$(3.11) \quad s_i^{(t+1)} = \zeta(W_{i,s}h^{(t)} + B_{i,s}) \in \mathcal{W}^* \cong \mathbb{R}^5$$

$$(3.12) \quad q_2^{(t+1)} = \zeta(W_q h^{(t)} + B_q) \in \mathcal{W}^* \cong \mathbb{R}^5,$$

$$(3.13) \quad e_i^{(t+1)} = \sigma(W_{i,e}h^{(t)} + B_{i,e}) \in \text{End}_{\mathbb{R}}(\mathcal{V}_i),$$

$$(3.14) \quad a_i^{(t+1)} = \sigma(W_{i,a}h^{(t)} + B_{i,a}) \in \mathcal{V}_i,$$

$$(3.15) \quad w_i^{(t+1)} = \sum_{i=0}^{N-1} s^{(t+1)}(\bar{i}) \cdot (R^*)^i(w_i^{(t)}),$$

$$(3.16) \quad r_1^{(t+1)} = \sum_{j=0}^L M_2^{(t)}(r_2^{(t)})_j \cdot R^j(r_1^{(t)}),$$

$$(3.17) \quad r_2^{(t+1)} = \sum_{i=0}^{N-1} q_2^{(t+1)}(\bar{i}) \cdot R^i(r_2^{(t)}),$$

$$(3.18) \quad M_i^{(t+1)} = (1_{\mathcal{V}_i} - e_i^{(t+1)}) \circ M_i^{(t)} + w_i^{(t)} \otimes a_i^{(t+1)}.$$

Finally, the evolution equation is

$$(3.19) \quad h_0^{(t+1)} = \sigma\left(V(M_1^{(t)}(r_1^{(t)})) + H_0 h_0^{(t)} + U_0 x^{(t+1)} + B_0\right).$$

The main difference between the ordinary NTM and the pattern NTM is in (3.16) 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.

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 an example in which, in addition to the original memory ring from the standard NTM, there are two rings with coefficients in $\llbracket \mathbf{int}_W \rrbracket$ and a ring with coefficients in $\llbracket \mathbf{bint}_W \rrbracket$. 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 $\mathbf{a} = \{a, b, \dots, z\} \cup \{\circ, \bullet, S\}$ and $\mathcal{S} = \mathcal{V}_1 = \bigoplus_{i \in \mathbf{a}} \mathbb{R}i$ 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

$$(3.20) \quad \circ abcdeSabcdeSabcde \bullet \longmapsto abcde aabccdde abcde,$$

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 $\llbracket m \rrbracket, \llbracket n \rrbracket$ respectively and the read address of the fourth ring is focused at $\llbracket S \rrbracket$ 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 $\llbracket 0 \rrbracket$ on the fourth ring we will follow the pattern on the second ring, and if we see $\llbracket 1 \rrbracket$ we will follow the pattern on the third ring. An entry $\llbracket 00 \rrbracket$ 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

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

with linear lambda term

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

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 $\llbracket \text{master}^{\text{out}} \rrbracket_{nl}(g)(h)$ now looks like

$$(3.21) \quad \llbracket \text{feed} \rrbracket (M_2(r_2), M_3(r_3), M_4(r_4))_{nl}(R)(r_1) .$$

To be more concrete, suppose the read address of the fourth ring is sharply focused at a position containing $\llbracket S \rrbracket$ where S contains a copies of 0 and b copies of 1. Suppose moreover that $\mathcal{V}_2 = \mathcal{V}_3 = \text{span}(\llbracket 0 \rrbracket, \dots, \llbracket L \rrbracket)$. Then at any given time the controller observes some linear combination of the contents of the second and third memory rings

$$(3.22) \quad M_2(r_2) = \sum_{i=0}^L \lambda_i \llbracket i \rrbracket, \quad M_3(r_3) = \sum_{j=0}^L \mu_j \llbracket j \rrbracket .$$

In this case the term in (3.21) is

$$(3.23) \quad \left(\sum_i \lambda_i R^i \right)^a \left(\sum_j \mu_j R^j \right)^b (r_1)$$

which can be expanded as

$$(3.24) \quad \sum_{u_0+\dots+u_L=a} \sum_{v_0+\dots+v_L=b} \frac{a!}{u_0! \dots u_L!} \frac{b!}{v_0! \dots v_L!} \lambda_0^{u_0} \dots \lambda_L^{u_L} \mu_0^{v_0} \dots \mu_L^{v_L} R^{\sum_t t(u_t+v_t)}(r_1).$$

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

Remark 3.6. 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.

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 (4.2) 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 $\llbracket 1 \rrbracket$ at every position of the second memory ring and the repeating pattern $\llbracket 0 \rrbracket, \llbracket 1 \rrbracket$ on both the third and fourth rings (note that in the first case these are of type **int**_W and in the second case of type **bint**_W).
2. The controller writes the input sequence $a, b, c, d, e, S, \dots, \bullet$ 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 $\llbracket 0 \rrbracket$ 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 $\llbracket 1 \rrbracket, \llbracket 1 \rrbracket, \dots$ 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 $\llbracket 0 \rrbracket, \llbracket 1 \rrbracket, \llbracket 0 \rrbracket, \dots$ 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 $V(a), V(a), V(b), V(b), \dots$
7. Finally, when the controller emits the vector $V(\bullet)$ we stop.

Remark 3.7. Following the notation of earlier sections, we write $r_i^{(t)}$ for the read address of the i -th memory ring, and $M_i^{(t)} \in \text{Hom}_{\mathbb{R}}(\mathcal{W}_i, \mathcal{V}_i)$ for the state of the memory. Then the analogue of (3.16) in the current model is, using the polynomials f_p from above,

$$r_1^{(t+1)} = \sum_p f_p(M_2^{(t)}(r_2^{(t)})_0, \dots, M_2^{(t)}(r_2^{(t)})_L, M_3^{(t)}(r_3^{(t)})_0, \dots, M_3^{(t)}(r_3^{(t)})_L) \cdot R^p(r_1^{(t)}).$$

The update equations for r_2, r_3, M_2, M_3 are as before.

4 Modal sequence NTM

For our final example extends the multiple pattern NTM with new modal symbols, which modify the way the two patterns are interpreted. We extend the alphabet to

$$(4.1) \quad \mathfrak{a} = \{a, b, \dots, z\} \cup \{\circ, \bullet, S\} \cup \{1, 2\}$$

with $\mathcal{S} = \mathcal{V}_1 = \bigoplus_{i \in \mathfrak{a}} \mathbb{R}i$ as before. The symbol S still switches the pattern used to read from the first memory ring, while the digits 1, 2 are modal operators that modify the way the dominant pattern is interpreted. To be more precise: they are modal in the sense that the last digit encountered reading from left to right is active at any given time, and they modify the pattern by multiplying (with 1, 2 respectively) the size of the steps taken when following the pattern in reading from the first memory ring. For example:

$$(4.2) \quad \circ 1abcde2Sabcd eSabcd e1abcde \bullet \mapsto abcde aaccee b b d d acebd abcde.$$

This can be achieved in the LLRNN framework by adding a memory ring with coefficients in $\llbracket \mathbf{int}_W \multimap \mathbf{int}_W \rrbracket$. To summarise, we now have five memory rings with coefficient spaces

$$\mathcal{V}_1 = \bigoplus_{i \in \mathfrak{a}} \mathbb{R}i, \quad \mathcal{V}_2 = \mathcal{V}_3 \subseteq \llbracket \mathbf{int}_W \rrbracket, \quad \mathcal{V}_4 \subseteq \llbracket \mathbf{bint}_W \rrbracket, \quad \mathcal{V}_5 \subseteq \llbracket \mathbf{int}_W \multimap \mathbf{int}_W \rrbracket.$$

Take $\mathcal{V}_2 = \mathcal{V}_3$ as in the previous section, and

$$\mathcal{V}_5 = \text{span}(\llbracket \underline{\text{mult}}(1, -) \rrbracket, \llbracket \underline{\text{mult}}(2, -) \rrbracket)$$

in the notation of [22, §6.1]. The relevant part of the master algorithm is

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

with linear lambda term

$$\underline{\text{newfeed}} = (\lambda p q r s a. ((s (((\underline{\text{feed}} p) q) r)) a)).$$

This appears in $\llbracket \text{master}^{\text{out}} \rrbracket_{nl}(g)(h)$ in the form

$$(4.3) \quad \llbracket \text{newfeed} \rrbracket(M_2(r_2), M_3(r_3), M_4(r_4), M_5(r_5))_{nl}(R)(r_1).$$

To give a concrete example: keep the notation of (3.22) and assume in addition that $M_5(r_5) = \sum_{c=1}^2 \gamma_c \llbracket \text{mult}(c, -) \rrbracket$. As before we assume for simplicity that the fourth ring is focused at $\llbracket S \rrbracket$ where S contains a copies of 0 and b copies of 1. Then (4.3) is

$$(4.4) \quad \sum_c \gamma_c \left(\left(\sum_i \lambda_i R^i \right)^a \left(\sum_j \mu_j R^j \right)^b \right)^c (r_1).$$

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 \mathcal{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 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

²See the “gradients” method of tensorflow/python/ops/gradients.py in TensorFlow 0.10

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 3.2 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 $V(S)$.

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, *Begriffsschrift, 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):179-211, 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. Mellies, *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.