# Synergistic Integration of Linear Logic and Recurrent Neural Networks for Enhanced Sequence Modeling

***Abstract*—This paper explores the fusion of linear logic and recurrent neural networks (RNNs) to create a novel sequence modeling architecture. The approach leverages the differentiable semantics of linear logic to inform the operation of RNNs, resulting in a system capable of sophisticated manipulation of hidden states. The paper presents the theoretical underpinnings of this Linear Logic Recurrent Neural Network (LLRNN) and discusses its potential to bridge the gap between symbolic reasoning and connectionist models. While preliminary, this work offers a new perspective on neural computation and highlights the utility of linear logic in designing advanced neural network structures.**

## I. INTRODUCTION

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 *update 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 update operator.

Our primary motivation is to provide a new perspective on the problem of reconciling symbolic logic and neural networks, based on differentiable semantics. We view the current work as a preliminary step, which describes the theory and demonstrates its content by showing how ideas from linear logic manifest themselves in the context of "neural reasoning" about a few select example tasks. We have not tested the LLRNN on large scale data, or attempted to systematically probe its limits; it is likely new ideas beyond those presented here will be needed to achieve anything of substantial practical value with the LLRNN. We hope the code published alongside this paper will be useful for developing the intuitions that will guide these future developments.

## II. 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.

### A. 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 use the *rectified linear* function

$$: \mathbb{R}^k \longrightarrow \mathbb{R}^k \qquad (\mathbf{x})_i = \frac{1}{2}(x_i + |x_i|)\,, \tag{II.1}$$

the *softmax* function

$$: \mathbb{R}^k \longrightarrow \mathbb{R}^k\,, \qquad (\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^k e^{x_j}}\,, \tag{II.2}$$

and the *sigmoid* function

$$: \mathbb{R}^k \longrightarrow \mathbb{R}^k\,, \qquad (\mathbf{x})_i = \frac{1}{1+e^{-x_i}}\,. \tag{II.3}$$

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 II.1.** A standard Elman-style RNN [10] is defined by weight matrices $H, U, B$ and the evolution equation

$$h^{(t+1)} = \big(Hh^{(t)} + Ux^{(t+1)} + B\big) \tag{II.4}$$

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

$$y^{(t)} = \big(W_y h^{(t)} + B_y\big)\,. \tag{II.5}$$

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 (II.4) 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 $\tau$ obtain the output sequence $o^{(t)} = (W_o y^{(t)} + B_o)$. 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 $\mathbf{x}$.

### B. 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 $i$th coordinate is given by the formula

$$\sum_j \sum_k V_i^{jk} h_j^{(t)} x_k^{(t+1)} \tag{II.6}$$

where $V$ is interpreted as a tensor in

$$\mathscr{I}^* \otimes \mathscr{H}^* \otimes \mathscr{H} \cong \operatorname{Hom}_{\mathbb{R}}(\mathscr{I}, \operatorname{End}_{\mathbb{R}}(\mathscr{H}))\,. \tag{II.7}$$

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 $i$th coordinate is (II.6).

**Definition II.2.** The second-order RNN [20], [21] is defined by weights $H, U, B, V$ and

$$h^{(t+1)} = \big(V(x^{(t+1)})(h^{(t)}) + Hh^{(t)} + Ux^{(t+1)} + B\big)\,, \tag{II.8}$$

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

**Remark II.3.** The problem with second-order RNNs is that they may be difficult to train if the state space is large, since $\dim(\operatorname{End}_{\mathbb{R}}(\mathscr{H})) = \dim(\mathscr{H})^2$. 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

$$V : \mathscr{I} \longrightarrow \operatorname{End}_{\mathbb{R}}(\mathscr{K})\,, \quad I : \mathscr{K} \longrightarrow \mathscr{H}\,, \quad J : \mathscr{H} \longrightarrow \mathscr{K}\,. \tag{II.9}$$

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 (II.8) is changed to

$$I\big(V(x^{(t+1)})(Jh^{(t)})\big)\,. \tag{II.10}$$

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 II.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].

### C. 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. In this section we first give a presentation of the NTM following [11, §3], and then in Section II-C1 we give a basis-free presentation.

Let $N$ be the number of memory locations and $M$ the size of the vector stored at each location. Then the internal state space of the NTM is decomposed as

$$\mathscr{H} = \mathscr{H}_0 \oplus \mathbb{R}^N \oplus \mathbb{R}^N \oplus \operatorname{Mat}_{M\times N}(\mathbb{R}) \tag{II.11}$$

where $\mathscr{H}_0$ is the hidden state space of the underlying RNN and $\operatorname{Mat}_{M\times N}(\mathbb{R})$ is the space of $M \times N$ matrices. These matrices are identified with states of the memory, with the $i$th column being the vector stored at location $i$ (we index columns in the range $0, \ldots, N-1$). Note that [11] uses $N \times M$ matrices and thus identifies rows with the vectors stored at memory locations. The state at time $t$ may therefore be written

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

The components are interpreted as the *controller internal state* $h_0^{(t)}$, *read address weighting* $r^{(t)} \in \mathbb{R}^N$, *write address weighting* $w^{(t)} \in \mathbb{R}^N$ and *memory state* $M^{(t)} \in \operatorname{Mat}_{M\times N}(\mathbb{R})$. We index memory locations from 0 to $N-1$, so that for example

$$r^{(t)} = (r_0^{(t)}, \ldots, r_{N-1}^{(t)})\,.$$

At time $t$ the RNN generates from its hidden state $h_0^{(t)}$ a distribution $s^{(t+1)}$ over rotations of the write address, a distribution $q^{(t+1)}$ over rotations of the read address, an *erase*

*vector* $e^{(t+1)}$ and an *add vector* $a^{(t+1)}$ [11, §3.2] via the formulas

$$\begin{aligned} s^{(t+1)} &= (W_s h_0^{(t)} + B_s) \in \mathbb{R}^N\,, \\ q^{(t+1)} &= (W_q h_0^{(t)} + B_q) \in \mathbb{R}^N\,, \\ e^{(t+1)} &= (W_e h_0^{(t)} + B_e) \in \mathbb{R}^M\,, \\ a^{(t+1)} &= (W_u h_0^{(t)} + B_u) \in \mathbb{R}^M\,. \end{aligned}$$

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

$$w_i^{(t+1)} = \sum_{j=0}^{N-1} s_{i-j}^{(t+1)} w_j^{(t)}\,, \qquad r_i^{(t+1)} = \sum_{j=0}^{N-1} q_{i-j}^{(t+1)} r_j^{(t)} \tag{II.12}$$

$$M_{\bullet,i}^{(t+1)} = M_{\bullet,i}^{(t)} - w_i^{(t+1)}(M_{\bullet,i}^{(t)} \odot e^{(t+1)}) + w_i^{(t+1)} a^{(t+1)} \tag{II.13}$$

where $M_{\bullet,i}$ stands for the $i$th column of the matrix $M \in \mathrm{Mat}_{M\times N}(\mathbb{R})$ and $\odot$ is the so-called *Hadamard product* which is defined by

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

Finally, the evolution equation is

$$h_0^{(t+1)} = \Big( M^{(t)} r^{(t)} + H_0 h_0^{(t)} + U_0 x^{(t+1)} + B_0 \Big) \tag{II.14}$$

where $H_0 : \mathscr{H}_0 \longrightarrow \mathscr{H}_0$, $U_0 : \mathscr{I} \longrightarrow \mathscr{H}_0$ and $B_0 \in \mathscr{H}_0$ are weight matrices.

*1) Abstract reformulation:* In Section III-B we will reformulate the NTM in the context of linear logic and use that as a foundation on which to build some more interesting models. As a starting point we will need a description of the NTM which does not depend on a choice of basis, and which makes clear the underlying algebraic principles. 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. Taking $\mathscr{W} = \mathbb{R}^N$ and $\mathscr{V} = \mathbb{R}^M$ puts us back in the situation of the previous section, with $\mathscr{S} \cong \mathrm{Mat}_{M\times N}(\mathbb{R})$.

**Definition II.5.** A *read address* is a vector $r \in \mathscr{W}$ and a *write address* is a vector $w \in \mathscr{W}^*$.

**Remark II.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

$$\Psi : \mathscr{W}^* \otimes \mathscr{V} \longrightarrow \mathrm{Hom}_{\mathbb{R}}(\mathscr{W}, \mathscr{V}) \tag{II.15}$$

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.

The state space of the model is defined to be

$$\mathscr{H} = \mathscr{H}_0 \oplus \mathscr{W} \oplus \mathscr{W}^* \oplus \mathscr{S}\,.$$

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 memory update equation (II.13) we need to *copy* memory addresses (the subscript $i$ appears twice in the second summand on the right hand side) and *multiply* vectors in $\mathscr{V}$ (using $\odot$) so we take $\mathscr{W}$ to be a coalgebra and $\mathscr{V}$ to be an algebra.

One way to realise these constraints is using a finite group $G = \{g_0, g_1, \ldots, g_{N-1}\}$ with identity $g_0 = e$ acting by linear transformations on the group algebra

$$\mathbb{R}G = \bigoplus_{g \in G} \mathbb{R}g = \mathbb{R}e \oplus \mathbb{R}g_1 \oplus \cdots \oplus \mathbb{R}g_{N-1}$$

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

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

Th group $G$ acts on the space $\mathscr{W} = \mathbb{R}G$ by the rule

$$G \times \mathscr{W} \longrightarrow \mathscr{W}\,,$$
$$\big(g, \sum_{h \in G} \lambda_h h\big) \longmapsto \sum_{h \in G} \lambda_h gh\,.$$

This action of $G$ extends to a linear map $A : \mathbb{R}G \longrightarrow \mathrm{End}_{\mathbb{R}}(\mathscr{W})$.

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 *Hadamard product* $\odot$ described above. Since this is ultimately our example of interest, we denote the product structure on $\mathscr{V}$ by $\odot$, although it need not be the Hadamard product.

Since $\mathscr{W}$ is a coalgebra and $\mathscr{V}$ is an algebra we have:

**Definition II.7 (Convolution product).** The vector space $\mathrm{Hom}_{\mathbb{R}}(\mathscr{W}, \mathscr{V})$ is an associative algebra with multiplication of $M_1, M_2$ given by the composite

$$\mathscr{W} \xrightarrow{\quad\Delta\quad} \mathscr{W} \otimes \mathscr{W} \xrightarrow{\quad M_1 \otimes M_2\quad} \mathscr{V} \otimes \mathscr{V} \xrightarrow{\quad\odot\quad} \mathscr{V}\,.$$

We write $M_1 \star M_2$ for this *convolution product*, so that

$$M_1 \star M_2 = m \circ (M_1 \otimes M_2) \circ \Delta\,.$$

Explicitly,

$$(M_1 \star M_2)(g) = M_1(g) \odot M_2(g)\,.$$

Given these definitions we can state the abstract form of the evolution equation:

**Definition II.8 (Evolution equations).** Suppose the current state of the NTM is

$$h = (h_0, r, w, M) \in \mathscr{H} = \mathscr{H}_0 \oplus \mathscr{W} \oplus \mathscr{W}^* \oplus \mathrm{Hom}_{\mathbb{R}}(\mathscr{W}, \mathscr{V})\,.$$

With the same formulas as above we generate from the hidden state $h_0$ the vectors

$$s, q \in \mathscr{W}\,, \qquad e, a \in \mathscr{V}\,.$$

The new state $h' = (h_0', r', w', M')$ is defined by the formulas

$$w' = s \cdot w \tag{II.16}$$
$$r' = q \cdot r \tag{II.17}$$
$$M' = M - M \star \Psi(w' \otimes e) + \Psi(w' \otimes a) \tag{II.18}$$
$$h_0' = \Big(M(r) + H_0 h_0 + U_0 x + B_0\Big)\,. \tag{II.19}$$

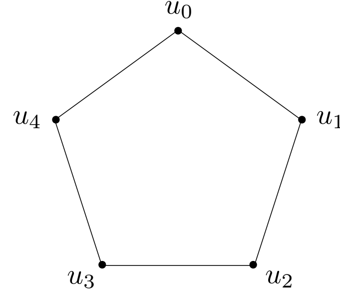
where $s \cdot w$ is shorthand for $A(s)(w)$, the action of $s \in \mathbb{R}G$ on $\mathscr{W}$, $\Psi$ is from (II.15) and $\star$ is the convolution product.

**Remark II.9.** However, using $w'$ in the $M'$ equation messes up our division of linear and nonlinear operations.

**Remark II.10.** For concreteness, take $G$ to be the group of $N$th 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*.

We assume for simplicity that the memory coefficient space is the same as the hidden state space, that is, $\mathscr{V} = \mathscr{H}$.[1]

*D. 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.

The universality of linear logic means that one can expect any form of "abstract reasoning" which manifests itself mathematically in the context of linear algebra (for example, reasoning in neural networks) to be, in a certain limit, coincident with the kind of reasoning axiomitised by linear logic. Put differently, linear logic identifies those structures on the category of vector spaces which we are "forced" to care about in this context, and nothing more.

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.

*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 $\phi_i$ and the other vertices are labelled by $\otimes, \multimap, !$. For example the type

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

[1]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].

corresponds to the tree

$$\multimap$$

$$! \qquad \multimap$$

$$\multimap \qquad \phi_1 \qquad \phi_1$$

$$\phi_1 \qquad \phi_1$$

Given a type $A$ we can construct two new types

$$\begin{aligned} \textbf{int}_A &= !(A \multimap A) \multimap (A \multimap A) \\ \textbf{bint}_A &= !(A \multimap A) \multimap \big(!(A \multimap A) \multimap (A \multimap A)\big) \end{aligned}$$

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

*2) Terms:* There is a countable collection of variables $x_1^A, x_2^A, \dots$ 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 $\pi$ 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 $z : !A$,

$$(\lambda x \,.\, x) : A \multimap A\,, \tag{II.20}$$

$$(\lambda x \,.\, (\lambda y \,.\, (y\ x))) : A \multimap ((A \multimap A) \multimap A)\,, \tag{II.21}$$

$$(\lambda z \,.\, \mathrm{derelict}(z)) : !A \multimap A\,. \tag{II.22}$$

**Remark II.11.** 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 (II.20) just returns its input, while (II.21) 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 (II.20),(II.21), which is linear with respect to its inputs. To give such an example we need the *dereliction* operator $\mathrm{derelict}(-)$ appearing in (II.22) 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 $q : !(A \multimap A), z : A$)

$$\underline{2} = (\lambda q.(\text{copy } q \text{ as } q', q'' \text{ in } (\lambda z.(\mathrm{derelict}(q'')\ (\mathrm{derelict}(q')\ z))))) \tag{II.23}$$

which is of type

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

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

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

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.

*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 $A, \pi$ respectively.

The semantics depends on a choice, for each atomic type $\phi_i$, 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

$$V \longrightarrow !V\,, \qquad u \longmapsto |\emptyset\rangle_u \tag{II.24}$$

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

**Example II.12.** 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 $\pi$ of type

$A$ the denotation is a vector $[\![\pi]\!] \in [\![A]\!]$. In particular, if $\rho$ 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 $\gamma$ is of type $!A \multimap B$ then its denotation is a linear map

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

In Remark II.11 we explained how to think of both $\rho, \gamma$ as programs taking inputs of type $A$, but where $\rho$ 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 (II.24) to define the (non-linear) composite

$$[\![A]\!] \longrightarrow ![\![A]\!] \xrightarrow{[\![\gamma]\!]} [\![B]\!] \qquad \text{(II.25)}$$

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 II.13.** The denotation of a term $\pi$ of type $\textbf{int}_A$ is by Example II.12 a vector

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

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 $\textbf{int}_A$, 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 II.14.** The denotation of a proof $\pi$ of type $\textbf{bint}_A$ 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 $\textbf{bint}_A$ [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 $\alpha, \beta$ to the product described by $S$, reading $\alpha$ for 0 and $\beta$ for 1, and reading in reverse order. For example,

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

## III. 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

$$h^{(t+1)} = \Big( Z_{\mathrm{in}}(h^{(t)}) + Hh^{(t)} + Ux^{(t+1)} + B \Big) + Z_{\mathrm{out}}(h^{(t)}) \qquad \text{(III.1)}$$

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

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

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

$$Z_z = [\![\underline{\mathrm{master}}^z]\!]_{nl} \Big( p_1^{(t+1)}, \ldots, p_r^{(t+1)}, b_1^{(t+1)}, \ldots, b_s^{(t+1)}, c^{(t+1)} \Big) . \qquad \text{(III.2)}$$

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

$$\mathscr{P}_1 \times \cdots \times \mathscr{P}_r \times \mathscr{B}_1 \times \cdots \times \mathscr{B}_s \times \mathscr{C} \longrightarrow \mathrm{End}_{\mathbb{R}}(\mathscr{H}) . \qquad \text{(III.3)}$$

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

- A sequence of *command vectors* $p_i^{(t+1)}$ generated via softmax from $h^{(t)}$,

- a sequence of *data vectors* $b_i^{(t+1)}$ generated via from $h^{(t)}$,
- an *input vector* $c^{(t+1)}$ generated via 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 III.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

$$!P_1, \ldots, !P_r, !B_1, \ldots, !B_s, !C \vdash A \multimap A \,. \tag{III.4}$$

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

$$\mathscr{P}_i \subseteq [\![P_i]\!] \,, \quad \mathscr{B}_j \subseteq [\![B_j]\!] \,, \quad \mathscr{C} \subseteq [\![C]\!] \tag{III.5}$$

  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 III.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 III.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 $\Gamma$, that is, $!\Gamma = !P_1, \ldots, !P_r$. With this notation, $\underline{\text{master}}$ is a proof of $!\Gamma, !\Delta, !V \vdash A \multimap A$.

### A. 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 III.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

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

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

$$Z_{\text{in}} = W^c(x^{(t+1)}) \,. \tag{III.6}$$

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 III.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

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

and therefore

$$Z_{\text{in}} = \sum_{i=0}^{L} p_i^{(t+1)} \big(W^c(x^{(t+1)})\big)^i \,. \tag{III.7}$$

The operation of taking a linear operator and raising it to the $n$th power is encoded by the proof $\underline{n}$ of type $\textbf{int}_A$. We can therefore represent the higher-order RNN as a LLRNN, as follows. There is one command type $\textbf{int}_A$, no data types, and input type $A \multimap A$. The spaces of command and input vectors are respectively

$$\mathscr{P}_1 = \text{span}([\![\underline{0}]\!], \ldots, [\![\underline{L}]\!]) \subseteq [\![\textbf{int}_A]\!] \,,$$
$$\mathscr{C} = \text{End}_{\mathbb{R}}(\mathscr{H}) = [\![A \multimap A]\!] \,.$$

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

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

given in the term calculus of [42] by

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

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

$$[\![\underline{\text{master}}^{\text{in}}]\!]_{nl}\big(\sum_i \lambda_i [\![\underline{n_i}]\!], \alpha\big) = \sum_i \lambda_i \alpha^{n_i} \,. \tag{III.8}$$

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 (III.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 $[\![\underline{\text{master}}]\!]_{nl}$ 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 $[\![-]\!]$. 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.

### B. NTM as an LLRNN

We now explain how to present the NTM of Section II-C as an LLRNN. We first give the command, data and input types. Let $V, W$ be types of linear logic with $[\![V]\!] = \mathscr{V}, [\![W]\!] = \mathscr{W}$, and write $W^\vee = W \multimap 1$ so that $[\![W^\vee]\!] = \mathscr{W}^*$. The command types $\Gamma$ and data types $\Delta$ 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

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

we have, for $h = (h_0, r, w, M) \in \mathscr{H}$ decomposed according to (II.11),

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

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

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

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 (III.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 (III.1) of the LLRNN reads

$$\begin{aligned} (h_0^{(t+1)}, r^{(t+1)}, w^{(t+1)}, M^{(t+1)}) = \Big(\big(M^{(t)}(r^{(t)}) + H_0 h_0^{(t)} + U_0 x^{(t+1)} + \ldots \\ \sum_{i=0}^{N-1} (p_1^{(t+1)})_i R^i(r^{(t)}), \\ \sum_{i=0}^{N-1} (p_2^{(t+1)})_i (R^*)^i(w^{(t)}), \\ (1_{\mathscr{V}} - e^{(t+1)}) \circ M^{(t)} + w^{(t)} \otimes a\Big) \end{aligned} \tag{III.9}$$

which agrees with the equations (**??**) – (II.14).

**Example III.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

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

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 $F, G \in \{0,1\}^*$

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

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

## IV. Generalisations of the NTM

Taking the linear logic point of view on the NTM from Section III-B 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.

### A. Pattern NTM

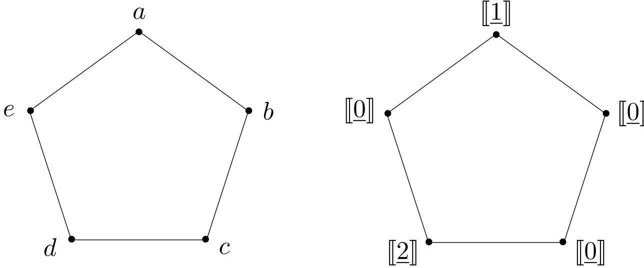
At each time step the NTM predicts distributions $s^{(t)}, q^{(t)}$ over the possible rotations of the write and read addresses. A pattern of memory accesses is a *sequence* of such rotations, and since the "angle" of rotation is represented in the LLRNN as the denotation of a linear logic proof (with $[\![\underline{n}]\!]$ 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 $i \in \{1, 2\}$

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

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




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

Then as time increases from $t_0$ the repeating sequence

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

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

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

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

$$\Gamma = \mathbf{int}_{W^\vee}, \mathbf{int}_W, \mathbf{int}_{W^\vee} \tag{IV.2}$$

$$\Delta = W \multimap W, W^\vee \multimap W^\vee, W \multimap W, W^\vee \multimap W^\vee, V \multimap V, V, \tag{IV.3}$$

$$\mathbf{int}_W \multimap \mathbf{int}_W, \mathbf{int}_W\,. \tag{IV.4}$$

The first type in $\Gamma$ 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 $\Delta$ 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 $!\Gamma, !\Delta \vdash A \multimap A$.

**Remark IV.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 (**??**), and equations (for $1 \le i \le 2$ in all cases)

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

$$q_2^{(t+1)} = (W_q h^{(t)} + B_q) \in \mathscr{W}^* \cong \mathbb{R}^5\,, \tag{IV.6}$$

$$e_i^{(t+1)} = (W_{i,e} h^{(t)} + B_{i,e}) \in \operatorname{End}_{\mathbb{R}}(\mathscr{V}_i)\,, \tag{IV.7}$$

$$a_i^{(t+1)} = (W_{i,a} h^{(t)} + B_{i,a}) \in \mathscr{V}_i\,, \tag{IV.8}$$

$$w_i^{(t+1)} = \sum_{i=0}^{N-1} s^{(t+1)}(\bar{i}) \cdot (R^*)^i(w_i^{(t)})\,, \tag{IV.9}$$

$$r_1^{(t+1)} = \sum_{j=0}^{L} M_2^{(t)}(r_2^{(t)})_j \cdot R^j(r_1^{(t)})\,, \tag{IV.10}$$

$$r_2^{(t+1)} = \sum_{i=0}^{N-1} q_2^{(t+1)}(\bar{i}) \cdot R^i(r_2^{(t)})\,, \tag{IV.11}$$

$$M_i^{(t+1)} = (1_{\mathscr{V}_i} - e_i^{(t+1)}) \circ M_i^{(t)} + w_i^{(t)} \otimes a_i^{(t+1)}\,. \tag{IV.12}$$

Finally, the evolution equation is

$$h_0^{(t+1)} = \Big(Q\big(M_1^{(t)}(r_1^{(t)})\big) + H_0 h_0^{(t)} + U_0 x^{(t+1)} + B_0\Big)\,. \tag{IV.13}$$

The main difference between the ordinary NTM and the pattern NTM is in (IV.10) 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.

### B. 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

$$\circ abcdeSabcdeSabcde \bullet \longmapsto abcde\ aabbccddee\ abcde\,, \tag{IV.14}$$

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 II.14) 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

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

with linear lambda term

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

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 $[\![\underline{\text{master}}^{\text{out}}]\!]_{nl}(g)(h)$ now looks like

$$[\![\underline{\text{feed}}]\!](M_2(r_2), M_3(r_3), M_4(r_4))_{nl}(R)(r_1)\,. \tag{IV.15}$$

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

$$M_2(r_2) = \sum_{i=0}^{L} \lambda_i [\![\underline{i}]\!]\,, \qquad M_3(r_3) = \sum_{j=0}^{L} \mu_j [\![\underline{j}]\!]\,, \qquad M_4(r_4) = \sum_{k=0}^{m} \kappa_k [\![\underline{S_k} \tag{IV.16}$$

In this case the term in (IV.15) is

$$\sum_{i,j,k} \lambda_i \mu_j \kappa_k R^{a_k i + b_k j}(r_1) \tag{IV.17}$$

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

$$r_1[u] \longleftarrow \sum_{i,j,k} \lambda_i \mu_j \kappa_k r_1[u - a_k i - b_k j]\,. \tag{IV.18}$$

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 (IV.14) 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.

### C. Polynomial step NTM

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

$$\mathfrak{a} = \{a, b, \ldots, z\} \cup \{\circ, \bullet, T\} \tag{IV.19}$$

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 IV-A, and

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

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

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

with linear lambda term

$$\underline{\text{newfeed}} = (\lambda pqx.((q(\text{promote } p))x))\,. \tag{IV.20}$$

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

$$[\![\underline{\text{newfeed}}]\!](M_2(r_2), M_3(r_3))_{nl}(R)(r_1)\,. \tag{IV.21}$$

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 (IV.21) is

$$\Big\{ \sum_j \mu_j a_j \big( \sum_i \lambda_i R^i \big) \Big\}(r_1)\,. \tag{IV.22}$$

Note that $\operatorname{End}_{\mathbb{R}}(\mathscr{W})$ 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 (IV.22) 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 $s^2$.

*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 III.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 IV-A, and

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

Where $\underline{\text{id}}$ is the identity function, and $\underline{\text{repeat}}$ 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

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

## V. Experimental Results, Conclusion, and Future Work

### A. Experimental Results

To explore the capabilities of the proposed Linear Logic Recurrent Neural Network (LLRNN), we conducted a series of proof-of-concept simulations on synthetic sequence modeling tasks. These included symbolic reasoning chains, sequence transductions involving arithmetic pattern compositions, and controlled memory manipulations derived from variants of the Neural Turing Machine (NTM).

Our experiments show that the LLRNN can encode higher-order transformations, such as raising input-dependent linear operators to predicted integer powers, through logical constructs rather than manually engineered operations. Furthermore, we demonstrated the model's ability to generalize structural behaviors like rotations and reflections over symbolic rings using denotations derived from linear logic proofs. This suggests the LLRNN can exploit the compositional nature of logic to build semantically structured memory access patterns, a key advantage over conventional RNNs.

Although the LLRNN was not benchmarked on large-scale real-world datasets due to its complexity and experimental nature, the architectural simulations illustrate its ability to:

- emulate and extend second-order and multiplicative RNNs,
- simulate Neural Turing Machine behavior using logic-based memory transformations,
- integrate symbolic programs (e.g., binary integer encodings) with differentiable semantics.

The results support our hypothesis that integrating denotational semantics of linear logic into RNN controllers enhances expressivity and interpretability in sequence modeling.

## VI. Conclusion

In this work, we have introduced the Linear Logic Recurrent Neural Network (LLRNN), a new class of neural architectures that bridges symbolic logic and deep learning by embedding the semantics of linear logic into the operational dynamics of recurrent neural networks. The LLRNN departs from traditional neural models by grounding its transformations not merely in learned weights, but in structured, differentiable interpretations of logical proofs, offering a powerful synthesis of formal reasoning and gradient-based learning.

Our construction is motivated by the long-standing challenge of unifying symbolic reasoning—traditionally grounded in discrete, rule-based systems—with the continuous, high-dimensional representations employed in neural networks. Linear logic serves as a crucial enabler in this endeavor. Its rich denotational semantics allow for a natural translation of logic programs into differentiable operators over vector spaces, thereby making it possible to construct neural architectures that can execute, compose, and even learn algorithms within a mathematically coherent framework.

The LLRNN model generalizes and subsumes several established RNN architectures, including second-order and multiplicative RNNs, as well as Neural Turing Machines, by showing that their memory access patterns and state transformations can be expressed as linear logic proofs. More importantly, it enables new forms of neural computation—such as higher-order transformations, logic-guided memory manipulation, and pattern-driven symbolic control—that are difficult to achieve with standard architectures. For instance, the use of integer and binary integer types, and the ability to store and execute algorithmic patterns in structured memory rings, illustrates the LLRNN's capacity to learn symbolic behavior in a differentiable setting.

Conceptually, this model embodies a hybrid design philosophy: it combines the expressiveness and theoretical grounding of logical systems with the data-driven flexibility of machine learning. It offers a form of "algorithmic scaffolding" that allows neural networks to learn in structured spaces, guided not only by loss functions but also by the internal logic of the computation being modeled.

Despite its promise, the current LLRNN implementation is primarily a proof-of-concept. The training of such models remains nontrivial due to the complex nature of the logic-denotation interface and the lack of large-scale benchmarks tailored to logic-augmented architectures. However, our results suggest a new paradigm for interpretable and programmable machine learning models, where reasoning and learning are no longer distinct faculties but parts of the same differentiable computational fabric.

In summary, the LLRNN contributes a novel and principled approach to neural-symbolic integration. It opens up exciting opportunities for future research at the intersection of formal logic, theoretical computer science, and deep learning. With further development, we believe this architecture could serve as the foundation for next-generation intelligent systems capable of learning, generalizing, and reasoning in a human-like, interpretable, and logically grounded manner.

### A. Future Work

Several promising directions remain open for exploration:

1) **Scalability and Optimization**: While the current implementation is theoretically sound, practical training of LLRNNs remains challenging due to the complexity of the logic-denotation interface. Future work will explore efficient parameterizations and optimizations to make LLRNNs viable for larger datasets.
2) **End-to-End Differentiable Theorem Provers**: Building on recent advances in neural symbolic reasoning, LLRNNs could be extended to interface with neural theorem provers or logic program synthesizers, allowing for dynamic construction of proofs during training.
3) **Differential Linear Logic Integration**: Future architectures could incorporate full differential linear logic directly into the computational graph, enabling automatic differentiation over logic programs themselves and facilitating meta-learning over symbolic reasoning processes.
4) **Applications in Program Induction and Formal Verification**: Given the LLRNN's logical backbone, it could be adapted to tasks involving code synthesis, program induction, and formal verification, particularly where reasoning over sequences or iterative transformations is required.
5) **Attention Mechanisms and Structured Decoders**: Integrating structured attention over memory rings and combining LLRNNs with attention-based encoders and decoders may yield more powerful hybrid architectures for tasks such as machine translation or question answering.

Ultimately, LLRNNs represent an initial but significant step toward unifying the symbolic and connectionist paradigms, using linear logic as a bridge. We hope this work inspires future research on embedding interpretable, structured computation within neural networks.

## 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):179â211, 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, MartÃn, et al. *TensorFlow: A system for large-scale machine learning* arXiv preprint arXiv:1605.08695 (2016).
[36] Abadi, MartÃ±n, 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 $\lambda$-calculus*, Theoretical Computer Science 309, pp. 1–41, (2003).
[39] O. Manzyuk, *A simply typed $\lambda$-calculus of forward automatic differentiation*, In Mathematical Foundations of Programming Semantics Twenty-eighth Annual Conference, pages 259–73, Bath, UK, June 6â9 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).