Linear logic and recurrent neural networks

Huiyi Hu, Daniel Murfet

January 25, 2017

1 Introduction

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

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

We demonstrate one concrete realisation of these ideas in the form of a Recurrent Neural Network (RNN) controller [6] augmented with spaces of linear logic programs. At each time step the controller predicts which commands (denotations of linear logic programs) to run and the inputs to run them on, from its hidden state and the current input. We call the resulting system the *Linear Logic Recurrent Neural Network* or LLRNN. This architecture is inspired by many papers in the neural network literature, most notably the second-order RNN [21] and multiplicative RNN [12].

2 Architecture

2.1 The second-order RNN

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

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

and by ζ the *softmax* function

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

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

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

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

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

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

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

We consider a generalisation of the second-order RNN [16, 17, 14, 15] and the similar multiplicative RNN [12]. In addition to the hidden-to-hidden matrix H and the input-to-hidden matrix U of the traditional RNN, the second-order RNN learns a matrix V that maps inputs to linear operators on the hidden state. More precisely, a vector is added to the evolution equation whose ith coordinate is given by the formula

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

where V is interpreted as a tensor in

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2 The linear logic RNN

At each time step the second-order RNN generates from its input a linear operator which is applied to to the hidden state; the operator is moreover a *linear* function of the input. The LLRNN model is similar to the second-order RNN in the sense that it generates at each time step a linear operator which is applied to the hidden state, but dissimilar in that this operator is a *non-linear* function of its arguments, which include the current input but also vectors predicted from the hidden state.

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

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

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

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

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

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

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

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

- a type A with $[\![A]\!] = \mathscr{H}$, the hidden state space;
- a proof $\underline{\text{master}}$ of the sequent

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

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

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

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

From this data we obtain the following function by restriction:

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

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

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

and the evolution equation

$$h^{(t+1)} = \sigma\Big(Z + Hh^{(t)} + Ux^{(t+1)} + B\Big) \tag{2.11}$$

where the new term Z is

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

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

Proposition 2.5. *The function* $[\![\underline{\text{master}}]\!]_{nl}$ *is smooth.*

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

Moreover, the derivatives of $[\![\underline{\text{master}}]\!]_{nl}$ 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.

2.3 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 [18, §7]. From this point of view, the purpose of augmenting an RNN with linear logic is to provide access to iterators of a complexity "similar" to the function that the neural network is trying to approximate.

The most elementary form of iteration is repetition, which in the current context means raising a linear operator to an integer power. This non-linear transformation of an input operator $\alpha \in \operatorname{End}_{\mathbb{R}}(\mathscr{H})$ to the output operator α^n for some integer n is encoded by a linear logic program (which is a synonym here for *proof*) of the type

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

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

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

as will be explained below. For more details on the following examples see [30, §3].

Example 2.6. The denotation of a proof π of type $\mathbf{int}_A$ is a function

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

which is a polynomial function of the entries of the input matrix. For example, with $\underline{n}$ for $n \geq 0$ the Church numeral of type $\mathbf{int}_A$, we have $[\![\underline{n}]\!]_{nl}(X) = X^n$.

Example 2.7. The denotation of a proof π of type $\mathbf{bint}_A$ is a polynomial function

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

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

$$[\![\underline{01101}]\!](X,Y) = YXYYX\,.$$

We begin with an example that makes only trivial use of the coupling to linear logic.

Example 2.8 (Second-order RNN). With $r = s = 0$ there are no command or data vectors. Take for the input type $C = A \multimap A$ and

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

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

$$Z = W^c(x^{(t+1)})(h^{(t)})\,. \tag{2.12}$$

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 2.9 (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 $h^{(t)}$ and the evolution equation is

$$Z = \sum_{i=0}^{L} p_i^{(t+1)} \big(W^c(x^{(t+1)})\big)^i (h^{(t)})\,. \tag{2.13}$$

The operation of taking a given linear operator and raising it to the nth power is what is encoded by the linear logic program $\underline{n}$ of type $\mathbf{int}_A$. So we can rephrase the higher-order RNN as a LLRNN by taking a single command vector (so $r = 1$), no data vectors, and an input type $C = A \multimap A$ as in the second-order RNN. The command type is $P_1 = \mathbf{int}_A$, and the space of command vectors is

$$\mathscr{P}_1 = \operatorname{span}([\![\underline{0}]\!], \ldots, [\![\underline{L}]\!]) \subseteq [\![\mathbf{int}_A]\!]\,.$$

We view $p^{(t+1)}$ as being a distribution over this space $\mathscr{P}$ of (denotations of) linear logic programs, and take the master algorithm to be the proof of the sequent

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

with the property that for $n \geq 0$ and $\alpha \in \operatorname{End}_{\mathbb{R}}(\mathscr{H})$,

$$[\![\underline{\text{master}}]\!]_{nl}\big([\![\underline{n}]\!], \alpha\big) = \alpha^n\,. \tag{2.14}$$

In this case $\underline{\text{master}}$ is just the evaluation program, which applies the given Church numeral to the given linear operator. By construction the LLRNN with this configuration reproduces the evolution equation with Z as in (2.13), so it is the higher-order RNN.

The raising of linear operators to a power predicted from the hidden state is the crucial architectural feature behind the location-based addressing scheme of the Neural Turing Machine (NTM) [7], Differentiable Neural Computer [1] and stack-augmented RNN [21]. The linear operator R involved in the case of the NTM is the *rotation* by one position of the memory state, viewed as a sequence of $N+1$ vectors arranged around a circle. The read and write weightings of the NTM are controlled by predicting a distribution, at each time step, over the N distinct powers $L^0 = I, L, L^2, \dots, L^{N-1}$ of this rotation matrix, and applying the linear combination of these powers to the vector representing the weighting.

Example 2.10 (Neural Turing Machine). The NTM with only location-based memory addressing can be realised in the LLRNN framework as follows. The *address space* is

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

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

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

The state of the memory $M^{(t)}$ at time t is a linear map $M^{(t)} : \mathscr{W} \longrightarrow \mathscr{V}$ which we view as an arrangement of the vectors $u_j = M^{(t)}(\bar{j})$ around the circle $\{e^{\frac{2\pi\sqrt{-1}j}{N}}\}_{j \in \mathbb{Z}/N\mathbb{Z}} \subseteq \mathbb{C}$, with u_j placed at position j. In the notation of [7] we may take $\mathscr{W} = \mathbb{R}^N$ and $\mathscr{V} = \mathbb{R}^M$ so that elements of $\mathscr{S}$ are naturally identified with $N \times M$ matrices.

The space of memory states has a natural action of the cyclic group of order N generated by "rotating the circle". Intuitively, if a memory state assigns the vector $u_j \in \mathscr{V}$ to position j around the circle for $0 \le j \le N-1$ then the rotated state assigns the vector u_{j+1} to the position j, with $j+1$ computed modulo N. Said differently, let $R : \mathscr{W} \longrightarrow \mathscr{W}$ be the linear map defined by

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

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

The internal state space of the RNN is decomposed as

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

and the state of the controller at time t is written

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

The components of this vector are interpreted as the *RNN controller internal state* $h_0^{(t)}$, *read address weights* $r^{(t)}$, *write address weights* $w^{(t)}$ and *memory contents* $M^{(t)}$. The linear maps H, U are constrained so as to be determined by linear maps $H_0 : \mathscr{H}_0 \longrightarrow \mathscr{H}_0$ and $U_0 : \mathscr{I} \longrightarrow \mathscr{H}_0$ and there is a new weight matrix $V : \mathscr{V} \longrightarrow \mathscr{H}_0$.

With ζ denoting softmax, we have

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

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

$$\begin{aligned}
w^{(t+1)} &= \sum_{i=0}^{N-1} s^{(t+1)}(\bar{i}) \cdot (R^*)^i(w^{(t)})\,,\\
r^{(t+1)} &= \sum_{i=0}^{N-1} q^{(t+1)}(\bar{i}) \cdot R^i(r^{(t)})\,,\\
M^{(t+1)} &= M^{(t)} + w^{(t+1)} \otimes u^{(t+1)}\,.
\end{aligned}$$

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

$$h^{(t+1)} = \sigma\Big(V\big(M^{(t+1)}(r^{(t+1)})\big) + H_0 h_0^{(t)} + U_0 x^{(t+1)} + B\Big)\,. \tag{2.15}$$

We omit the sharpening step [7, Eq. (9)] for simplicity. The original NTM paper [7] is not specific about how the output $M^{(t+1)}(r^{(t+1)})$ of the "read heads" enters the RNN evolution equation; the form we have given above follows the construction of the differentiable neural computer [1, p. 7], which is a refinement of the NTM.

We now explain how to recover the NTM with location-based addressing as a special case of the LLRNN. Let V, W be types with $[\![V]\!] = \mathscr{V}, [\![W]\!] = \mathscr{W}$, and write $W^\vee = W \multimap 1$ so that $[\![W^\vee]\!] = \mathscr{W}^*$. The command types are $\mathbf{int}_W, \mathbf{int}_{W^\vee}$ (proofs of which respectively define iterators on operators on $\mathscr{W}$ and $\mathscr{W}^*$) and the data types are $W \multimap W, W^\vee \multimap W^\vee$.

We take the master program to be the proof of

$$!\mathbf{int}_W, !\mathbf{int}_{W^\vee}, !(W \multimap W), !(W^\vee \multimap W^\vee), !V \vdash A \multimap A$$

such that for $(h, r, w, M) \in \mathscr{H}$,

$$\begin{aligned}
&[\![\underline{\text{master}}]\!]_{nl}\big([\![\underline{m}]\!], [\![\underline{n}]\!], \alpha, \beta, u\big)(h, r, w, M)\\
&\qquad = \big(M(r) + \beta^n(w)\big(\alpha^m(r)\big) \cdot u, \alpha^m(r), \beta^n(w), M + \beta^n(w) \otimes u\big)\,.
\end{aligned}$$

We take $\mathscr{B}_1 = \mathrm{End}_{\mathbb{R}}(\mathscr{W})$, $\mathscr{B}_2 = \mathrm{End}_{\mathbb{R}}(\mathscr{W}^*)$ and

$$\begin{aligned}
\mathscr{P}_1 &= \mathrm{span}([\![\underline{0}]\!], \ldots, [\![\underline{N-1}]\!]) \subseteq [\![\mathbf{int}_W]\!]\,,\\
\mathscr{P}_2 &= \mathrm{span}([\![\underline{0}]\!], \ldots, [\![\underline{N-1}]\!]) \subseteq [\![\mathbf{int}_{W^\vee}]\!]
\end{aligned}$$

so that as in the example of the higher-order RNN, the command vectors $p_1^{(t+1)}, p_2^{(t+1)}$ give distributions over the set of integers $\{0, \ldots, N-1\}$ which are to be used as exponents of the given operators α, β. Fixing these operators to be $\alpha = R, \beta = R^*$ recovers the NTM.

Remark 2.11. If we take instead the coefficient vector space of the memory to be $\mathrm{End}_{\mathbb{R}}(\mathscr{H})$ so that we are storing operators rather than vectors, and change the RNN update equation to have the term

$$\big(M^{(t+1)}(r^{(t+1)})\big)(h_0^{(t)})$$

then the NTM is a generalisation of the higher-order RNN (which is the special case where $\mathscr{W} = \mathbb{R}$, i.e. $N = 1$). Then all the generalisations we have in mind are very sound, conceptually. Test.

Remark 2.12. A more powerful system would substitute a neural theorem prover along the lines of [26, 28] in place of the libraries of functional programs $\mathscr{P}_i$. At each time step the RNN controller would predict a continuous vector, which when fed into the neural theorem prover as a set of parameters, generates a symbolic program whose denotation is then coupled back into the RNN [27].

Remark 2.13. In functional programming languages like differential λ-calculus [34] and differential linear logic [29] it is possible to differentiate programs with respect to their inputs, even if the programs are higher-order (that is, take functions as input and return them as output). This is a generalisation of automatic differentiation [33] which is widely used to compute derivatives of complicated real-valued functions, for example in the backpropagation algorithms of machine learning packages like TensorFlow [31, §4.1]. The idea is to augment every computation in a code fragment so that derivatives with respect to a chosen variable are computed along with the main result. In TensorFlow this is done by adding special nodes in the dataflow graph.[1] The idea of differential λ-calculus is similar, but more complex [35]. It would be interesting to explore augmenting the dataflow graph of TensorFlow directly with terms of differential linear logic, in a way that generalises the coupling between semantics and RNNs in this paper.

3 Background on CPUs

Recall that an assembly program for an ordinary CPU looks like

```
LOAD R1, A
ADD R3, R1, R2
STORE C, R3
```

Where `R1`,`R2`,`R3` stand for the first three registers of the CPU and `A`,`B`,`C` are numbers representing addresses in memory. Thus series of instructions will result in the CPU fetching a number from memory location `A` and storing it in `R1`, adding this number to a previously stored number in `R2` with the result being stored in `R3`, and finally writing that register out to the memory address `C`. In the analogy between a CPU and a vanilla

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

RNN we think of the number read from `A` as the current input $x^{(t)}$ and the previously stored value in `R2` as (part of) the internal state $h^{(t-1)}$.

Recent work [?, ?] on coupling memory to neural networks takes as its starting point the first of the above instructions `LOAD R1, A` and makes it "differentiable" by having the RNN controller predict at time t both the memory address `A` and the register `R3` to write to (in this case for example, as a mask on the vector giving the internal state $h^{(t+1)}$). The same differentiable interpretation can be given of the `STORE` command. This is done by adding suitable terms to the update equation (2.1).

In contrast our focus is on the third command, the `ADD`. We increase the expressive power of the update equation by allowing it to predict at time t an operation $p^{(t)}$ (a vector specifying a point in a space of "programs") which is to be performed on the input and internal state. In order to preserve our analogy with CPU instructions even without `LOAD` and `STORE`, we could imagine a CPU with a command

```
ADD R3, A, R2
```

which at once reads the number from address `A`, adds it to the stored value in `R2` and writes the result to `R3`. Note that without a `LOAD` instruction, the only way a value could have gotten into `R2` in a previous time step is as the result of another `ADD`.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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