Linear logic and recurrent neural networks

Huiyi Hu, Daniel Murfet

January 24, 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 [4]. If we follow Leibniz [1] 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 [6], stack-augmented recurrent neural networks [21] and others [8, 22, 23, 27]. In this paper we investigate a new conceptual approach, based on the Curry-Howard isomorphism and linear logic.

The Curry-Howard isomorphism [3] 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, 2, 30].

We demonstrate one concrete realisation of these ideas in the form of an recurrent neural network (RNN) controller [5] which predicts at time t a linear combination of denotations of linear logic programs (or proofs) to be applied to vectors which are themselves predicted from the hidden state of the RNN at time $t-1$ and the input symbol at time t. The output of the linear logic program is a linear operator, which is applied to the hidden state at time $t-1$ to generate a vector which is then added to the usual update equation of the RNN. This architecture borrows its basic features from various sources, including the second-order RNN [21], neural machine translation system [?], matrix-vector models in natural language processing [?] and the multiplicative RNN [?]. We apply it to the classical problem [?] of approximating a function $f : A^* \longrightarrow A^*$ of sequences in some alphabet A.

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 [5] 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

$$y^{(t)} = \sigma\big(Lh^{(t)} + M\big) \,. \tag{2.2}$$

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 $o^{(t)} = \tau(Ey^{(t)} + F)$. 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 [15, 16, 13, 14] and the similar multiplicative RNN [11]. 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 [15, 16] 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 [11] 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 [11] and sentiment analysis [10]. This is not to be confused with the *multiplicative integration* RNN of [9] 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 [10].

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 Linear Logic Recurrent Neural Network (LLRNN) generalises this by generating at each time step a sequence of *commands* (which are linear combinations of denotations of linear logic programs) which are applied to a sequence of *input operators* by a fixed *master algorithm*. The commands depend only on the previous hidden state, while the input operators depend only on the inputs $x^{(t+1)}, x^{(t)}, \ldots, x^{(t-s+2)}$ for some *window length* s.

We refer to [17, 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 [17, §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 [17, 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 <u>master</u> of the sequent

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

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

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

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

From this data we obtain the following function by restriction:

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

The coupling of the master algorithm to the RNN controller is by weight matrices, which are linear maps

$$W_i : \mathscr{H} \longrightarrow \mathscr{P}_i, \qquad V_j : \mathscr{I} \longrightarrow \mathscr{B}_j \,.$$

Finally, the evolution equation is given by

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

where the new term Z is

$$Z = \llbracket \underline{\text{master}} \rrbracket_{nl}\Big(W_1(h^{(t)}), \ldots, W_r(h^{(t)}), V_1(x^{(t+1)}), V_2(x^{(t)}), \ldots, V_s(x^{(t-s+2)}) \Big)(h^{(t)}) \,.$$

Remark 2.5. TODO: actually the inputs to the master algorithm should also have nonlinearities. We should just have B_i instead of $B_i \multimap B_i$ and the difference is that the commands have a softmax and the others have a ReLU. Well but in the case of memories we should allow there to be ReLU predictions from the hidden.

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

Proposition 2.6. *The function* $\llbracket \underline{\text{master}} \rrbracket_{nl}$ *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 $\llbracket \underline{\text{master}} \rrbracket_{nl}$ can be computed symbolically using the cut-elimination algorithm of differential linear logic, and it is therefore feasible to implement a general LLRNN in a software package like TensorFlow.

As defined the LLRNN is a class of models, depending on the choices of the command and input types, the master algorithm, and the spaces of command vectors and input operators. We will now give several examples of LLRNN models defined by various choices of these parameters. For an introduction to linear logic see [37, 17] and for more details on the following examples see [30, §3]. The types we will use are

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

called respectively the types of *integers* on A, and *binary integers* on A.

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

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

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

Example 2.8. The denotation of a proof π of type $\mathbf{bint}_A$ is a polynomial function

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

For every binary sequence $S \in \{0,1\}^*$ 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,

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

Remark 2.9. The subspaces $\mathscr{P}_i$ should be viewed as chosen "libraries" of denotations of linear logic programs, that the RNN controller is able to call. For example, if $P_i = \mathbf{bint}_A$ then we might take for $\mathscr{P}_i$ the subspace spanned by $[\![\underline{01}]\!]$, $[\![\underline{110}]\!]$.

Example 2.10 (Second-order RNN). With $r = 0, s = 1$ and $B_1 = A$ take the master algorithm to be the proof of $!(A \multimap A) \vdash A \multimap A$ which is given by dereliction, so that with $\mathscr{B}_1 = \mathrm{End}_{\mathbb{R}}(\mathscr{H})$ we have $[\![\underline{\text{master}}]\!]_{nl}$ equal to the identity map on $\mathrm{End}_{\mathbb{R}}(\mathscr{H})$ and

$$Z = V_1(x^{(t+1)})(h^{(t)})\,.$$

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

Example 2.11 (Higher-order RNN). With $r = 1, s = 1$ we 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 $\alpha \in \mathrm{End}_{\mathbb{R}}(\mathscr{H})$,

$$[\![\underline{\text{master}}]\!]_{nl}\big([\![\underline{n}]\!], \alpha\big) = \alpha^n\,. \tag{2.12}$$

That is, the master evaluates the given Church numeral on the given linear operator. We take $\mathscr{P} = \mathrm{span}([\![\underline{1}]\!], \ldots, [\![\underline{L}]\!])$ for some integer L and $\mathscr{B} = \mathrm{End}_{\mathbb{R}}(\mathscr{H})$. Then the LLRNN learns a linear map $W : \mathscr{H} \longrightarrow \mathscr{P}$, whose value on a vector h we denote

$$W(h) = \sum_{i=1}^{L} W(h)_i [\![\underline{i}]\!]$$

and a linear trasformation $V : \mathscr{I} \longrightarrow \mathrm{End}_{\mathbb{R}}(\mathscr{H})$ as in the second-order RNN. Then

$$\begin{aligned} Z &= [\![\underline{\text{master}}]\!]_{nl}\big(W(h^{(t)}), V(x^{(t+1)})\big)(h^{(t)}) \\ &= \sum_{i=1}^{L} W(h^{(t)})_i V(x^{(t+1)})^i (h^{(t)})\,. \end{aligned}$$

When $L = 1$ this is just the second-order RNN.

Example 2.12 (Stack-augmented RNN). Thinking of an RNN as a state machine automata, and by analogy with a pushdown automata, an RNN can be augmented with an external stack memory [15]. We can emulate the more recent version of this idea in [21] with appropriate choices of the defining data of the LLRNN. We begin with a decomposition of the hidden state space as $\mathscr{H} = \mathscr{H}_0 \oplus \mathscr{S}$ where the "memory stack"

$$\mathscr{S} = k[z]/z^{N+1} = k1 \oplus kz \oplus \cdots \oplus kz^N \tag{2.13}$$

is equipped with the push and pop operators $p_+, p_- : \mathscr{S} \longrightarrow \mathscr{S}$ with matrices

$$p_+ = \begin{pmatrix} 0 & 0 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}, \qquad p_- = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}.$$

We also constrain the weight matrix H to be of the form

$$H = \begin{pmatrix} H_0 & 0 \\ 0 & 0 \end{pmatrix}$$

with respect to the direct sum decomposition $\mathscr{H} = \mathscr{H}_0 \oplus \mathscr{S}$.

The master program is a proof (which we omit) of

$$!1,\ !(\mathbf{int}_A \oplus \mathbf{int}_A),\ !(S \multimap S),\ !(S \multimap S) \vdash A \multimap A \tag{2.14}$$

with the property that $[\![S]\!] = \mathscr{S}$ and

$$[\![\underline{\text{master}}]\!]_{nl}(\tau, [\![\underline{m}]\!] \oplus [\![\underline{n}]\!], \alpha, \beta) = \tau J(1) + J \circ \alpha^m \circ \beta^n \circ I\,,$$

where $I : \mathscr{H} \longrightarrow \mathscr{S}$ and $J : \mathscr{S} \longrightarrow \mathscr{H}$ are the projections and injections respectively. We note that for $\lambda, \mu \in k$

$$[\![\underline{\text{master}}]\!]_{nl}(\tau, \lambda[\![\underline{1}]\!] \oplus [\![\underline{0}]\!] + \mu[\![\underline{0}]\!] \oplus [\![\underline{1}]\!], \alpha, \beta) = \tau J(1) + \lambda J \circ \alpha \circ I + \mu J \circ \beta \circ I\,.$$

The command types are $P_1 = 1, P_2 = \mathbf{int}_A \oplus \mathbf{int}_A$ and the input types are $B_1 = B_2 = S$. We take $\mathscr{B}_j = \mathrm{End}_{\mathbb{R}}(\mathscr{S})$ for $j \in \{1, 2\}$, $\mathscr{P}_1 = k$ and

$$\mathscr{P}_2 = \mathrm{span}([\![\underline{1}]\!] \oplus [\![\underline{0}]\!], [\![\underline{0}]\!] \oplus [\![\underline{1}]\!]) \subseteq [\![\mathbf{int}_A]\!] \oplus [\![\mathbf{int}_A]\!]\,. \tag{2.15}$$

We assume the weight matrices V_j are fixed during training with the image of every basis element equal to p_+ for $j = 1$ and p_- for $j = 2$.

Now let us explain why this emulates a stack-based memory. At time t, we write

$$h^{(t)} = (h_0^{(t)}, s^{(t)})^T$$

and view $h_0^{(t)}$ as the internal state of the RNN and $s^{(t)} = (s_0^{(t)}, \ldots, s_N^{(t)})^T$ as the contents of the stack at time t. Suppose that the command vectors generated by the controller are the scalar $W_1(h^{(t)}) \in k$ and

$$W_2(h^{(t)}) = \lambda [\![\underline{1}]\!] \oplus [\![\underline{0}]\!] + \mu [\![\underline{0}]\!] \oplus [\![\underline{1}]\!]$$

for some $\lambda, \mu \in k$. Then with $\alpha = p_+$ and $\beta = p_-$ and we have

$$\begin{aligned} Hh^{(t)} + Z &= J[\![\underline{\text{master}}]\!]_{nl}\big(W_1(h^{(t)}), W_2(h^{(t)}), p_+, p_-\big) I(h^{(t)}) \\ &= Hh^{(t)} + W_1(h^{(t)}) J(1) + J\big(\lambda p_+ + \mu p_-\big) I(h_0^{(t)}, s^{(t)})^T \\ &= Hh^{(t)} + J\left(W_1(h^{(t)}) + \mu s_1^{(t)}, \quad \lambda s_0^{(t)} + \mu s_2^{(t)}, \quad \cdots, \quad \lambda s_{N-1}^{(t)}\right)^T \\ &= \left(H_0 h_0^{(t)}, \quad W_1(h^{(t)}) + \mu s_1^{(t)}, \quad \lambda s_0^{(t)} + \mu s_2^{(t)}, \cdots, \lambda s_{N-1}^{(t)}\right) \end{aligned}$$

which agrees with the hidden state of the stack-augmented RNN [21] with hidden-to-hidden weight matrix H_0, after it has predicted with probability λ to push $W_1(h^{(t)})$ onto the stack, and with probability μ to pop the stack. To store vectors in a vector space V rather than a scalar at each stack position, we simply replace $\mathscr{S}$ by $\mathscr{S} \otimes V$.

Example 2.13 (Neural Turing Machine). A Neural Turing Machine (NTM) with only location-based memory addressing [6] can be realised in the LLRNN framework as follows. We introduce the *address space* $\mathscr{W} = \mathbb{Z}/N\mathbb{Z} \otimes_{\mathbb{Z}} \mathbb{R}$ and *memory coefficient space* $\mathscr{V}$ in terms of which the *memory space* is the tensor product

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

For example, in the notation of [6] we may take $\mathscr{W} = \mathbb{R}^N$ and $\mathscr{V} = \mathbb{R}^M$ so that "memory" tensors in $\mathscr{S}$ are naturally identified with $N \times M$ matrices.

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 $V : \mathscr{V} \longrightarrow \mathscr{H}_0$.

Let $R : \mathscr{W} \longrightarrow \mathscr{W}$ be the linear map defined by "rotation"

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

with dual map $R^* : \mathscr{W}^* \longrightarrow \mathscr{W}^*$. With f denoting softmax, we have

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

The update equation for the address [6, Eq. (8)] and memory [6, Eq. (3),(4)] are

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

Here we write $u^{(t)}$ for the difference $\mathbf{a}_t - \mathbf{e}_t$ of [6, §3.2]. Finally, the evolution equation of the NTM is given by

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

We omit the sharpening step [6, Eq. (9)] for simplicity.

In the LLRNN framework we take the master program to be a proof of

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

with $\mathscr{P}_1 = \mathscr{P}_2 = \operatorname{span}(\llbracket \underline{0} \rrbracket, \ldots, \llbracket \underline{N-1} \rrbracket)$ and $\llbracket W \rrbracket = \mathscr{W}, \llbracket W^\vee \rrbracket = \mathscr{W}^*, \llbracket V \rrbracket = \mathscr{V}$ such that

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

Remark 2.14. 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 [17, §7]. The most elementary forms of iteration are simply repetitions, as in the proofs of type $\mathbf{int}_\alpha, \mathbf{bint}_\alpha$ above. Our proposal is that in order to enable an RNN controller to learn functions of a certain complexity, one should provide it with access to iterators of a "similar level" of complexity.

Remark 2.15. 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 2.16. 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 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
```

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

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] 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

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

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

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

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

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

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

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

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

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

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

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

[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.