Linear logic and recurrent neural networks

Huiyi Hu, Daniel Murfet

December 10, 2016

1 Introduction

An old problem in artificial intelligence is *program induction*: given a set of input-output pairs, the problem is to produce a program which generalises the given pairs. There is a large literature on this problem [**?**, **?**] but it has received renewed attention in the past few years due to the advances in deep learning [**?**, **?**, **?**]. The general theme of much of this recent work has been to use an RNN or LSTM controllers coupled to additional computational elements, such as memory or a library of special functions.

Our approach is to augment an RNN controller with a space of functional programs based on the polynomial semantics of intuitionistic linear logic [**?**]. It is clear in principle that linear logic is well-suited for coupling to neural networks, and the aim of this paper is to demonstrate that in the simplest nontrivial case this coupling is actually of some practical value. Conceptually, one of the fundamental contributions of linear logic is that it demonstrates how to represent a general class of programs in terms of matrices, and multiplication of matrices.

2 Architecture

The architecture is a modified RNN. As usual we use "weight" as a synonym for variable, or more precisely, the variables which we will vary during gradient descent. At time $t \geq 1$ we denote the hidden state by $h^{(t)}$ and the input by $x^{(t)}$. We denote by σ the function

$$\sigma : \mathbb{R}^k \longrightarrow \mathbb{R}^k$$
$$\sigma(X)_i = \frac{1}{2}(X_i + |X_i|)\,.$$

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 $h^{(t)}$.

Definition 2.1. A standard Elman-style RNN [1] 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

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

Our augmented RNN adds a new term to the evolution equation, in the style of recent multiplicative RNNs [5, 7, 6]. The key intuition is that the evolution equation maps inputs to transformations of the internal state. To explain, 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. If our sequences are of a fixed length T then the RNN computes a function of input sequences

$$\mathbf{x} = \big(x^{(1)}, \ldots, x^{(T)}\big) \in \mathscr{I}^{\oplus T}\,.$$

The value of the RNN on the input $\mathbf{x}$ is computed by applying (2.1) for $0 \le t \le T-1$, 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

$$o^{(t)} = \tau(Ey^{(t)} + F)\,.$$

This is a sequence of vectors $\mathbf{o} = (o^{(1)}, \ldots, o^{(T)}) \in \mathscr{I}^{\oplus T}$ where 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 $\mathbf{x}$.

Note that the evolution equation implicitly determines a piecewise-linear function

$$\begin{gathered} \mathscr{I} \longrightarrow \operatorname{Func}(\mathscr{H}, \mathscr{H})\,, \\ x \mapsto \sigma\big(H(-) + Ux + B\big)\,, \end{gathered}$$

which specifies how we may view inputs as transforming the internal state of the RNN. One class of modifications of the RNN architecture adds more complicated (for example, piecewise-polynomial of higher degree) transformations of the internal state.

For example, compositional matrix-space models in NLP [6] add a new term to (2.1) built from an auxiliary learned linear map from the input space $\mathscr{I}$ into the space of linear operators on the state space $\operatorname{End}_{\mathbb{R}}(\mathscr{H})$. Our approach is similar, but we use a particular class of *non-linear* maps obtained as the denotations of proofs in linear logic. In principle proofs of any type could be coupled with an RNN controller, but here we consider only proofs of the following two types:

$$\begin{gathered} \mathbf{int}_\alpha = {!}(\alpha \multimap \alpha) \multimap (\alpha \multimap \alpha)\,, \\ \mathbf{bint}_\alpha = {!}(\alpha \multimap \alpha) \multimap \big({!}(\alpha \multimap \alpha) \multimap (\alpha \multimap \alpha)\big)\,. \end{gathered}$$

Under the Curry-Howard correspondence a proof π of type $\mathbf{bint}_\alpha$ can be thought of as a functional program taking two inputs of type $\alpha \multimap \alpha$ and giving an output of the same type. Conceptually proofs of type $\mathbf{int}_\alpha$ and $\mathbf{bint}_\alpha$ encode certain kinds of iteration. Rather than attempt to give a review of linear logic here, we give some examples.

Example 2.2. The *denotation* of a proof π of type $\mathbf{int}_\alpha$ is a function

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

which is a polynomial function of the entries of the input matrix. That is, writing $\mathscr{H} = \mathbb{R}^{n_H}$ and identifying $\operatorname{End}_{\mathbb{R}}(\mathscr{H})$ with $M_{n_H}(\mathbb{R})$ we have that $[\![\pi]\!](X)_{ij} : M_{n_H}(\mathbb{R}) \longrightarrow \mathbb{R}$ is a polynomial in the entries X_{kl} for $1 \leq i, j \leq n_H$. For every integer $n \geq 0$ there is a proof $\underline{n}$ of type $\mathbf{int}_\alpha$ called the *Church numeral*, whose denotation is the function $[\![\underline{n}]\!](X) = X^n$.

Example 2.3. The denotation of a proof π of type $\mathbf{bint}_\alpha$ is a polynomial function

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

For every binary sequence $S \in \{0,1\}^*$ there is a corresponding proof $\underline{S}$ of type $\mathbf{bint}_\alpha$, where $[\![\underline{S}]\!]$ sends a pair of matrices X, Y to the product described by S, reading X for 0 and Y for 1. For example,

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

An important insight, made especially clear in linear logic, is that the complexity of an algorithm computing a function on binary sequences $\{0,1\}^T \longrightarrow \{0,1\}^T$ has to do with the "level of iteration" that is used in the algorithm. The most elementary forms of iteration are simply repetitions, as in the proofs of type $\mathbf{int}_\alpha, \mathbf{bint}_\alpha$ above, but there are much more powerful forms of iteration which lead to more complex algorithms.

The intuition guiding this paper is that an RNN controller will have an easier time approximating functions of a certain complexity, if it has access to iterators (from linear logic) of a similar level of complexity. At the lowest level these iterators take the form of polynomial operations on matrices, since we are taking our cue from compositional matrix-space models by mapping input symbols to matrices acting on the hidden state. The abstract theory suggests how to couple more sophisticated iterators to an RNN, but our focus in this paper is on proving the simplest version of the concept.

Let $\Lambda = (\lambda_1, \ldots, \lambda_l)$ be a sequence of proofs of $\mathbf{int}_\alpha$ and let $\Pi = (\pi_1, \ldots, \pi_k)$ be proofs of $\mathbf{bint}_\alpha$. When we say that we allow the RNN controller to call this library of functional programs, what we actually mean is that it can predict a linear combination $p^{(t)}$ of the polynomial programs $[\![\lambda_i]\!]$ to be applied to its internal state, and similarly for the π_i.

Definition 2.4. The *linear logic* RNN coupled to Λ, Π has weight matrices H, U, B as in the standard RNN, plus new weights V, P, Q, C, D and an evolution equation:

$$\begin{aligned} p^{(t+1)} &= \sigma\big(Ph^{(t)} + C\big)\,, \\ q^{(t+1)} &= \sigma\big(Qh^{(t)} + D\big)\,, \\ h^{(t+1)} &= \sigma\Big(Z + Hh^{(t)} + Ux^{(t+1)} + B\Big) \end{aligned}$$

where

$$Z = \sum_{i=1}^{l} p_i^{(t+1)} [\![\lambda_i]\!](Vx^{(t+1)})\big(h^{(t)}\big) + \sum_{i=1}^{k} q_i^{(t+1)} [\![\pi_i]\!](Vx^{(t)}, Vx^{(t+1)})\big(h^{(t)}\big)\,.$$

The weight matrix V is interpreted as a linear map $\mathscr{I} \longrightarrow \mathrm{End}_{\mathbb{R}}(\mathscr{H})$ so that $Vx^{(t)}$ denotes a matrix of size $n_H \times n_H$ if $\mathscr{H} = \mathbb{R}^{n_h}$. The outputs $y^{(t)}$ of the linear logic RNN are defined by the same equation (2.2) as before.

Note that the RNN evolution equation for $h^{(t+1)}$ involves both the current input $x^{(t+1)}$ and the previous input $x^{(t)}$. We do this in order to allow the RNN to learn local transformations of an input sequence, for instance, the transformation of a sequence $a_0a_1a_2a_3\cdots$ which swaps every pair of symbols, to get $a_1a_0a_3a_2\cdots$ might use $[\![t_{10}]\!]$. More complicated transformations could be learned by adapting the equation to a function of $x^{(i)}, \ldots, x^{(t+1)}$ with a more complicated type than $\mathbf{bint}_\alpha$.

Example 2.5. If $\Lambda = \{\underline{1}, \underline{2}\}$ and $\Pi = \{\underline{010}\}$ then $p^{(t)} \in \mathbb{R}^2, q^{(t)} \in \mathbb{R}$ and

$$Z = p_1^{(t)}(Vx^{(t)})h^{(t)} + p_2^{(t)}(Vx^{(t)})^2 h^{(t)} + q^{(t)}(Vx^{(t)})(Vx^{(t+1)})(Vx^{(t)})h^{(t)}\,.$$

To be clear, here $(Vx^{(t)})h^{(t)}$ is a matrix multiplication. This term is identical to those in multiplicative RNNs [**?**].

Remark 2.6. If $\mathscr{I} = \mathbb{R}^{n_I}$ and $\mathscr{H} = \mathbb{R}^{n_H}$ then $V \in M_{n_H^2 \times n_I}(\mathbb{R})$. To avoid an explosion of weights due to the large integer n_H^2, we can map $\mathscr{I}$ to operators on a subspace of $\mathscr{H}$. This is done by choosing a decomposition $\mathscr{H} = \mathscr{H}_0 \oplus \mathscr{H}_1$ and letting $\rho : \mathscr{H} \longrightarrow \mathscr{H}_1$, $\iota : \mathscr{H}_1 \longrightarrow \mathscr{H}$ be respectively be the projection to and inclusion of the subspace $\mathscr{H}_1$. In the evolution equation we use $\iota[\![\lambda_i]\!](Vx^{(t+1)})\big(\rho h^{(t)}\big)$ with $[\![\lambda_i]\!]$ now a function $\mathrm{End}_{\mathbb{R}}(\mathscr{H}_1) \longrightarrow \mathrm{End}_{\mathbb{R}}(\mathscr{H}_1)$.

Remark 2.7. Here we give a comparison to other similar modifications of RNNs:

- The *second-order RNN* [11, 12, 9, 10] adds to the evolution equation a vector in $\mathscr{H}$ whose ith coordinate is given by the formula

  $$\sum_j \sum_k W_{ijk} h_j^{(t)} x_k^{(t+1)}. \tag{2.3}$$

  That is, W may be interpreted as a tensor in

  $$\mathscr{I}^* \otimes \mathscr{H}^* \otimes \mathscr{H} \cong \mathrm{Hom}_{\mathbb{R}}(\mathscr{I}, \mathrm{End}_{\mathbb{R}}(\mathscr{H}))\,,$$

  and the term we add to the evolution equation is $W(x^{(t+1)})(h^{(t)})$. Thus the second-order RNN learns a mapping of inputs to operators on its state space. The same equation (2.3) is referred to as the Tensor RNN in [7].

- The *multiplicative integration* RNN of [5] has a term

  $$Hh^{(t)} \odot Ux^{(t+1)} \tag{2.4}$$

  where $\odot$ is the Hadamard product, i.e. the entry-wise product.

- The Tensor RNN [7] takes as its intuition the idea a vector in $\mathscr{H}$ whose ith coordinate is

  $$\sum_j \sum_m (x^{(t+1)})_m W_{ijm} h_j^{(t)}$$

  where $W_{hh}^{(1)}, \ldots, W_{hh}^{(M)}$ is a sequence of matrices.

In [5] they claim the second-order RNN is hard to optimise [5, p.8], which makes our higher-order terms even scarier.

A 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. This could be used with a differentiable memory, but for the sake of simplicity we do not try this. 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.

Remark A.1. Although the architecture takes some inspiration from normal CPUs, there is an important distinction: on a normal CPU the program is given as a series of instructions prior to the beginning of execution. In contrast, in the model we have described each command is *predicted* at runtime from the current internal state. Perhaps we can understand the process intuitively as follows: we are co-learning a part of H, call it H_0, which generates some part of the internal state $h_0^{(1)}, h_0^{(2)}, \dots$ giving a path through the state space on which the weight matrix P picks out the right program to run at each time step. The overall algorithm is distributed amongst the weights of H_0 and P.

This also suggests an alternative algorithm: we do not predict $p^{(t)}$ at each time step, rather we have some fixed number of time steps T and matrices of weights $p^{(1)}, \dots, p^{(T)}$ which are learned by gradient descent.

B Old stuff

The operation to be performed is given by a sequence of vectors

$$p_i^{(t)} = \sigma(P_i h^{(t-1)}) \in \mathbb{R}^{n_P}\,, 1 \le i \le m \tag{B.1}$$

in a way that we will now explain. In outline, we think of the entries of $p_i^{(t)}$ as telling us the coefficients of monomials in the entries of $x^{(t)}$ and $h^{(t-1)}$ to use in the modified update equation. For simplicity let us only consider monomials of degree 2 in what follows, since the general case is similar. We set

$$y^{(t)} = Ex^{(t)} \oplus h^{(t-1)} \in \mathbb{R}^{n_Y} \tag{B.2}$$

where E is another matrix of weights. See Example **??** for an explanation.

Let F denote a linear map $F : M_{n_Y}(\mathbb{R}) \longrightarrow \mathbb{R}^{n_Y^2}$ which reads off the entries of a matrix into a column vector. In TensorFlow we can represent this using *reshape*. Writing $Y = y^{(t)}$ observe that YY^T is an $n_Y \times n_Y$ matrix with (i,j)-entry Y_iY_j. We choose $n_P = n_Y^2$ and compute the entry-wise multiplication

$$p_i^{(t)} \odot F(YY^T) = p_i^{(t)} \odot F\big(y^{(t)}(y^{(t)})^T\big) \in \mathbb{R}^{n_P}\,.$$

Finally, $q^{(t)}$ is the column vector whose ith row is $p_i^{(t)} \odot F(YY^T)$, that is,

$$q^{(t)} = \begin{pmatrix} p_1^{(t)} \odot F\big(y^{(t)}(y^{(t)})^T\big) \\ \vdots \\ p_m^{(t)} \odot F\big(y^{(t)}(y^{(t)})^T\big) \end{pmatrix}\,.$$

In summary, we view $x^{(t)}, h^{(t-1)}$ as respectively the differentiable analogues of `A`,`R2` and the sequence $p_1^{(t)}, \ldots, p_m^{(t)}$ as the analogue of the command `ADD`. The output of the command is the vector $q^{(t)}$. We incorporate this output into the update equation as follows:

$$h^{(t+1)} = \sigma\big(Vq^{(t)} + Hh^{(t)} + Ux^{(t+1)}\big)\,. \tag{B.3}$$

Thus V is the differentiable analogue of the register `R3`. The weights are P_i from (B.1), E from (B.2) and V, H, U from (B.3). This architecture is easily generalised to polynomials of higher degree, by adding additional terms.

Example B.1. Let us consider how the system might reproduce the program that repeats every digit of an input binary sequence, e.g.

$$0110 \longmapsto 00111100\,. \tag{B.4}$$

We take the inputs $x \in \mathbb{R}^2$ with $e_1 = (1,0)$ standing for the binary digit 1 and $e_0 = (0,1)$ standing for 0. We suppose that the system has learned the embedding matrix E such that $A = E(e_1)$ and $B = E(e_0)$ are matrices in $M_n(\mathbb{R}_{>0})$ with the property that the subgroup they generate under multiplication is a free group on two letters. This condition just means that the map

$$\Psi : \{0,1\}^* \longrightarrow M_n(\mathbb{R})$$

from binary sequences to matrices, defined inductively for $s \in \{0,1\}$ by

$$\Psi(sS) = \begin{cases} B\Psi(S) & s = 0 \\ A\Psi(S) & s = 1 \end{cases}$$

is injective. The space of matrices $\mathscr{H} = M_n(\mathbb{R})$ is the internal state of our RNN. To extract output from the RNN we apply a series of fully-connected layers with the final internal state $h^{(T)}$ as input, and we think of this series of layers as approximating a function $\Psi' : M_n(\mathbb{R}) \longrightarrow \{0,1\}^*$ with the property that $\Psi' \circ \Psi = 1$, that is, which can read off from a product of matrices ABA the corresponding binary sequence 101. So, in order to approximate the function (B.4) our RNN needs to take the inputs

$$x^{(1)} = B, x^{(2)} = A, x^{(3)} = A, x^{(4)} = B$$

and produce the final internal state

$$h^{(T)} = BBAAAABB \in M_n(\mathbb{R})\,.$$

This can be be done if we assume that in the update equation (B.3) has weights $H, U = 0$ and V is chosen so that

$$Vq^{(t)} = (x^{(t)})^2 h^{(t-1)}\,.$$

Note that the right hand side is a cubic polynomial in the entries of $x^{(t)}, h^{(t-1)}$ so we actually need the generalised form of (B.3).

References

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

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

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

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

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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