Linear logic and recurrent neural networks

Huiyi Hu, Daniel Murfet

December 6, 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.

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 $\mathbf{x}$ is computed by applying (2.1) for $0 \leq t \leq T-1$ to compute the vector $h^{(T)}$, and then feeding this into a second decoder neural network to output the final value [8]. For the moment we do not specify the decoder. The key point is that the evolution equation implicitly determines a piecewise-linear function

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

which specifies how we may view inputs as transforming the internal state of the RNN.

The key idea of compositional matrix-space models in Natural Language Processing (NLP) [**?**] is to augment the above mapping by learning an embedding of the input space into a space of matrices which acts on the internal state space. Our aim is to generalise this, based on proofs π in linear logic of type $\mathbf{bint}_\alpha$ where α is a variable whose denotation is $[\![\alpha]\!] = \mathscr{H}$. Such a proof has a presentation in the sequent calculus of the form

$$\begin{array}{c} \pi \\ \vdots \\ !(\alpha \multimap \alpha), !(\alpha \multimap \alpha) \vdash \alpha \multimap \alpha \end{array}$$

and its denotation $[\![\pi]\!]$ is a polynomial function

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

where $\mathrm{End}_{\mathbb{R}}(V)$ denotes the space of linear maps from a vector space V to itself.[1]

Let Π be a finite sequence of proofs $\Pi = (\pi_1, \ldots, \pi_k)$ of $\mathbf{bint}_\alpha$.

Definition 2.2. The *linear logic* RNN associated to Π has weight matrices H, U, B as in the standard RNN, together with new weight matrices V, Q, and an evolution equation

$$h^{(t+1)} = \sigma\Big(\sum_i Q_i [\![\pi_i]\!](Vx^{(t)}, Vx^{(t+1)})\big(h^{(t)}\big) + Hh^{(t)} + Ux^{(t+1)} + B \Big) . \tag{2.2}$$

Here the weight matrix V determines a linear map $V : \mathscr{I} \longrightarrow \mathrm{End}_{\mathbb{R}}(\mathscr{H})$ and $Q \in \mathbb{R}^k$.

[1] To explain what we mean by a "polynomial function", suppose $\mathscr{H} = \mathbb{R}^r$ so that $\mathrm{End}_{\mathbb{R}}(\mathscr{H})$ is the space of matrices $M_r(\mathbb{R})$. Then for all $1 \leq i, j \leq r$, the denotation $[\![\pi]\!]_{i,j}$ is a function of pairs of matrices $X, Y \in M_r(\mathbb{R})$ which is a polynomial in the entries X_{ab}, Y_{cd}.

Remark 2.3. To make it explicit, suppose $\mathscr{I} = \mathbb{R}^{n_I}$ and $\mathscr{H} = \mathbb{R}^{n_H}$, then $H \in M_{n_H \times n_H}(\mathbb{R})$, $U \in M_{n_H \times n_I}(\mathbb{R})$, $B \in M_{n_H \times 1}(\mathbb{R})$, $V \in M_{n_H^2 \times n_I}(\mathbb{R})$, $Q \in M_{k \times 1}(\mathbb{R})$. Since n_H^2 may be very large, we can try to act only on some manageable subspace of $\mathscr{H}$ via projections.

3 Examples

Example 3.1. Suppose $x^{(t)} = (a)$ and $h^{(t-1)} = (b)$ so that $n_Y = 2$ and

$$y^{(t)} = \begin{pmatrix} a & b \end{pmatrix}^T , \qquad YY^T = \begin{pmatrix} a^2 & ba \\ ab & b^2 \end{pmatrix} , \qquad F(YY^T) = \begin{pmatrix} a^2 & ba & ab & b^2 \end{pmatrix}^T .$$

If $p_1^{(t)} = \begin{pmatrix} 2 & -1 & 0 & 3 \end{pmatrix}^T$ and $p_2^{(t)} = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix}^T$ then

$$q^{(t)} = \begin{pmatrix} 2a^2 - ba + 3b^2 \\ a^2 \end{pmatrix} .$$

Example 3.2. Let us consider how the system might reproduce the program that repeats every digit of an input binary sequence, e.g.

$$0110 \longmapsto 00111100 . \tag{3.1}$$

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 (3.1) 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 (2.5) 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 (2.5).

A From Inputs to Operators

We understand how to build ordinary deep nets in semantics of linear logic if we add one more connective. Perhaps the right question to ask is: how to add RNNs, with shared weights? Well, that's clear. We are just iterating the map Φ, which we can represent as the promotion of one of the things we already understand.

This helps somewhat because we should only do things consistent with this picture, let it dictate the "right way" to do things.

Or we can just have $P = p^{(t)}$ not depend on time at all, nor on the previous stae. The main point is that the programs we can plug in are denotations of type

$$!\mathscr{H} \otimes \mathscr{I} \longrightarrow \mathscr{H}\,.$$

We probably don't understand enough examples right now to know whether these should be linear in $\mathscr{I}$. For example, we just map basis elements of $\mathscr{I}$ to particular matrices, which we may then choose to raise to some power.

Given vector spaces V, W let $\mathrm{Hom}(V, W)$ denote the space of linear maps and $\mathrm{Func}(V, W)$ the set of all functions. Let $\mathscr{H}$ denote the state space of the RNN, so that $h^{(t)} \in \mathscr{H}$ for all t. We view the evolution equation (2.1) of the ordinary RNN as a function defined on the space $\mathscr{I}$ of possible inputs

$$\begin{aligned}\Phi &: \mathscr{I} \longrightarrow \mathrm{Func}(\mathscr{H}, \mathscr{H})\,,\\ \Phi(x)(h) &= \sigma\big(Hh + Ux\big)\,.\end{aligned}$$

In this sense the RNN, viewed as a system of weights U, H, defines a mapping from *states* (the inputs) to *operators* on $\mathscr{H}$. From such a mapping we obtain an operator on $\mathscr{H}$ from any list of elements of $\mathscr{I}$ in the usual way:

Example A.1. The list type in System F is [**?**]

$$\mathrm{List}_U = \Pi X.(U \to X \to X) \to (X \to X)\,.$$

We can emulate the denotation of a binary sequence as follows: associate with X the space $\mathbb{R}^2$ and choose two matrices $A, B \in \mathbb{R}^k$. We consider an augmented RNN as above which, given the list $(0, 0, 1)$ outputs the associated matrix AAB. The idea is that e.g. at the last time step we should have

$$h^{(t)} = AB \oplus (\cdots)\,, x^{(t+1)} = (1, 0)$$

we need to produce A from $(1, 0)$ and then

$$h^{(t+1)} = Ah^{(t)}.$$

Example A.2. Do the same, but apply A^2 which means we use a cubic function of the inputs (since we need to square the A we get).

The intuition is that the input should *operate* on the internal state, and the particular operation may be a complicated function of the input we get. That is, we need to provide the state to operator correspondence. So our V term is the thing that provides this service. And which mapping to use depends itself on our internal state. That is, we need to predict the state to operator correspondence.

Maybe the linear logic connection is different. Maybe it goes via taking the linear logic denotations as inspiration for how to architect and think about neural networks processing data in particular ways (i.e. we use List to think about RNNs).

B Syntax

To the syntax of linear logic we add a connective which represents the nonlinearity (in our case, ReLU). It can be introduced on the right of the turnstile and is idempotent. It seems like once the inputs are all specified the value is either zero or the true value, so it acts a bit like a weird additive connective.

We can introduce the connective freely on the left of the turnstile. Cut corresponds to propagating the partitions back to the very left. Well maybe on the left of the turnstile we are only allowed to have a single formula (since we partition it). Also we probably only partition those inputs that are ! So we first have to have only !s on the left, like with promotion. But then instead of promotion we get a new connective.

Instead of forcing an NTM to learn a complicated series of things just to emulate iteration (which is going to be hard for it to do) we augment the "CPU" by adding an additional step (beyond arithmetic, reading and writing) where it can call out to some functional program that is chosen differentiably.

C Differentiable CPU

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 C.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.

D Old stuff

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

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

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

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

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:

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

Thus V is the differentiable analogue of the register `R3`. The weights are P_i from (2.3), E from (2.4) and V, H, U from (2.5). This architecture is easily generalised to polynomials of higher degree, by adding additional terms.

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.