# Linear logic and recurrent neural networks

Huiyi Hu, Daniel Murfet

November 29, 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.

Our approach is to augment an RNN controller with a space of functional programs, which can be called at each time step; in principle this should increase the expressive power of the overall system. Several prior works [?, ?] have considered making functional elements such as arithmetic available to the controller, but our approach is more general and is based on the development of a connection to linear logic, an intuitionistic logic which under the Curry-Howard correspondence may be viewed as a functional programming language with novel treatment of linear types. The denotational semantics of linear logic naturally involves linear algebra and polynomial matrices [?], making it a natural candidate for coupling to neural networks.

## 2 Details

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 $t$ we denote the hidden state by $h^{(t)}$ and the input by $x^{(t)}$. We denote by $\sigma$ the function

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

In a normal RNN we would have matrices $H, U$ of weights and define the time evolution of the hidden state by the equation

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

Our augmented RNN is defined by adding a new term to this equation. To motivate the modification, recall that one of the guiding principles of recent research is that an RNN or LSTM is analogous to a "neural CPU" with internal state $h^{(t)}$. The internal state of a CPU is just the contents of its registers.

**Remark 2.1.** 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`.

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

$$p_i^{(t)} = \sigma(P_i h^{(t-1)}) \in \mathbb{R}^{n_P}\,, 1 \le i \le m \tag{2.2}$$

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

$$y^{(t)} = Ex^{(t)} \oplus h^{(t-1)} \in \mathbb{R}^{n_Y} \tag{2.3}$$

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 $Y_i Y_j$. 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 $i$th 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:

$$h^{(t+1)} = \sigma\big(Vq^{(t)} + Hh^{(t)} + Ux^{(t+1)}\big) \,. \tag{2.4}$$

Thus $V$ is the differentiable analogue of the register `R3`. The weights are $P_i$ from (2.2), $E$ from (2.3) and $V, H, U$ from (2.4). This architecture is easily generalised to polynomials of higher degree, by adding additional terms.

**Example 2.2.** 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

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

**Remark 2.3.** 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 $h_0^{(1)}, h_0^{(2)}, \ldots$ 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 $p^{(1)}, \ldots, p^{(T)}$ which are learned by gradient descent.

**Example 2.4.** Let us consider how the system might reproduce the program that repeats every digit of an input binary sequence, e.g.

$$0110 \longmapsto 00111100\,. \tag{2.5}$$

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 (2.5) 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 (2.4) 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 (2.4).

## 3 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 $\Phi$, 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 $\operatorname{Hom}(V, W)$ denote the space of linear maps and $\operatorname{Func}(V, W)$ 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

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

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 3.1.** 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 3.2.** 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).

## 3.1 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.

## References