Linear logic and recurrent neural networks

Daniel Murfet

November 23, 2016

1 Introduction

One of the oldest problems in artificial intelligence is *program induction*: given a set of input-output pairs (say of binary sequences), 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 [**?**, **?**, **?**].

Arguably this problem is hard because learning methods are generally continuous, whereas programs (realised as say Turing machines, or λ-terms, or C programs, etc) are basically discrete. In this paper we initiate a novel two-step approach to the problem. In this first step, we couple a standard recurrent neural network to "differentiable programmatic components" which are vectors in a vector space which is the denotation of a type in linear logic. Once the optimisation algorithm has found a good vector representative of programs in this space, we apply the second step, which uses proof search to find a proof in linear logic whose denotation is observationally equivalent to the vector program.

2 Notes

We really only need to add a primitive for the nonlinearity, which we imagine as some sort of connective that can be introduced on the right of the turnstile. It is idempotent. What about elimination? 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.

- LSTMs and other modifications of RNNs seem to be exploiting higher degree polynomials. As does iteration in semantics of linear logic. One of our aims is to try and explore this connection (if any)

- simplest possible coupling of programs from linear logic to RNNs
- full space vs linear combination of known programs
- does it find the known programs if we give it the basic inputs?

The way an NTM is structured is like making a CPU differentiable. There are time steps, and at each time step a series of operations, such as arithmetic, reading from and writing to memory. But there are different approaches to making differentiable computing, depending on what level you want your model to make differentiable. For example, an NTM is not learning and algorithm but rather the operational semantics of an algorithm. Our approach is slightly closer to the algorithm itself, since we learn a denotation which can be executed by linear algebra.

Or one can view it as: 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