Linear logic and recurrent neural networks

Huiyi Hu, Daniel Murfet

December 18, 2016

1 Introduction

In this paper we investigate a natural approach to the synthesis of functional programming and neural networks, based on semantics of the former in differential categories. We apply this to the problem of *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 library of functional programs using the polynomial semantics of differential linear logic [13, 26]. Differential λ-calculus [30] and the related system of differential linear logic [25] may be viewed as functional programming languages in which programs naturally admit derivatives. These derivatives exist at both the syntactic and semantic level, and the differential structure of the language is what guarantees that the programs are compatible with backpropagation. The existence of this general theory demonstrates the naturality of our approach, but in the present paper the programs involved are so simple that we will not need to invoke the theory; for more details see Section 2.4.

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. 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

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

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 [11, 12, 9, 10] and the similar multiplicative RNN [7]. 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

$$\mathscr{I}^* \otimes \mathscr{H}^* \otimes \mathscr{H} \cong \mathrm{Hom}_{\mathbb{R}}(\mathscr{I}, \mathrm{End}_{\mathbb{R}}(\mathscr{H}))\,. \tag{2.4}$$

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 [11, 12] 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 $\dim(\mathrm{End}_{\mathbb{R}}(\mathscr{H})) = \dim(\mathscr{H})^2$. The *multiplicative RNN* is introduced

in [7] 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

$$V : \mathscr{I} \longrightarrow \mathrm{End}_{\mathbb{R}}(\mathscr{K}), \quad I : \mathscr{K} \longrightarrow \mathscr{H}, \quad J : \mathscr{H} \longrightarrow \mathscr{K}. \tag{2.6}$$

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 [7] and sentiment analysis [6]. This is not to be confused with the *multiplicative integation* RNN of [5] 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 [6].

2.2 Linear logic

In order to improve the expressive strength of an RNN, that is, allow it to approximate more complex functions from sequences of input symbols to sequences of output symbols, a natural idea is to provide a library of functional programs that can be "called" at each time step. The main problem lies in arranging for these programs to be differentiable, so that gradients can be backpropagated through them at training time.

One natural approach is to have an abstract, symbolic *source* language in which the relevant functional programs can be written, together with a concrete, differentiable *target* category in which the programs can be interpreted. This interpretation should respect the structure of the functional programs: for example, composition or plugging of programs should be interpreted as composition of differentiable maps. The purpose of the source language is to ensure expressive strength, that is, to provide a library of differentiable maps sufficiently rich to encode nontrivial symbolic transformations. The purpose of the target category is to facilitate a straightforward coupling into the RNN.

A candidate for the source language is linear logic [16] which has denotational semantics in differentiable maps [25]. We use the particular semantics developed in [13, 14, 26]. The second-order RNN encourages us to view symbols as operators on a vector space, and this aligns very well with the intuition provided by linear logic.

The *types* of intuitionistic linear logic (ILL) are built from atomic variables $\alpha, \beta, \ldots$ by applying binary connectives $\otimes, \multimap$ (tensor, Hom) and a unary connective ! (called bang). For example, for any type A we have the types

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

respectively of integers on A, and binary integers on A. The semantics will map A to a vector space, and $A \multimap A$ to the space of linear operators on that vector space.

A program π of type M, written $\pi : M$, is built from atomic pieces by rules that we will not explain here. The simplest rule is composition or *plugging*: given programs $M : \alpha \multimap \beta$ and $N : \beta \multimap \gamma$ we have a combined program $NM : \alpha \multimap \gamma$. A program of type $!A \multimap B$ takes inputs of type A and returns outputs of type B. The same is true of a program of type $A \multimap B$ but in addition it satisfies a *linearity constraint* in that it uses its input *precisely once*. So the ! connective marks inputs that may be reused freely (or not used at all). All inputs and outputs are themselves programs of the appropriate type.

Example 2.4. A program π of type $\mathbf{int}_A$ takes an input M of type $A \multimap A$ that may be used non-linearly, and outputs a program of type $A \multimap A$. For each integer $n \geq 0$ there is a program $\underline{n} : \mathbf{int}_A$ called the nth *Church numeral*, with the property that it maps an input M to the n-fold repetition M^n.

The Church numeral is an *iterator* (something like a *for* loop in imperative programming) which takes a program M and iterates it n times.

Example 2.5. A program π of type $\mathbf{bint}_\alpha$ maps an input of type $\alpha \multimap \alpha$ to a program of type $\mathbf{int}_\alpha = !(\alpha \multimap \alpha) \multimap (\alpha \multimap \alpha)$. So, given two programs M, N of type $\alpha \multimap \alpha$, we can apply the output $\pi(M)$ to N, to get a program of type $\pi(M)(N) : \alpha \multimap \alpha$.

Given a binary sequence $S \in \{0,1\}^*$ there is a program $\underline{S} : \mathbf{bint}_\alpha$ with the property that it maps an input pair (M, N) to the program which intertwines copies of M, N according to S. For example, $\underline{01101}$ sends (M, N) to the program $MNNMN$.

A *denotational semantics* of this language selects for each type A some mathematical object $[\![A]\!]$ and for a program of type $A \multimap B$ a transformation $[\![A]\!] \longrightarrow [\![B]\!]$ [21]. The precise kind of mathematical object, and transformations between them, is determined by the kind of category we take as the target of the semantics. In this paper we use the Sweedler semantics in vector spaces of [13, 14] and the polynomial semantics which refines it. We will not give the details, since we only need a small number of examples.

Starting with an atomic type α whose denotation $[\![A]\!] = \mathscr{K}$ is some finite-dimensional vector space over $\mathbb{R}$, we have

$$[\![A \multimap A]\!] = \mathrm{Hom}_{\mathbb{R}}([\![A]\!], [\![A]\!]) = \mathrm{End}_{\mathbb{R}}(\mathscr{K})\,.$$

Example 2.6. The denotation of a proof π of type $\mathbf{int}_\alpha$ is a function

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

which is a polynomial function of the entries of the input matrix. For example,

$$[\![\underline{n}]\!](X) = X^n\,.$$

To be clear, writing $\mathscr{K} = \mathbb{R}^{n_H}$ and identifying $\mathrm{End}_{\mathbb{R}}(\mathscr{K})$ with the space of matrices $M_{n_H}(\mathbb{R})$, the denotation of any proof of $\mathbf{int}_\alpha$ will be a function $[\![\pi]\!](X)_{ij} : M_{n_H}(\mathbb{R}) \longrightarrow \mathbb{R}$ which is a polynomial in the entries X_{kl} for $1 \leq i, j \leq n_H$.

Example 2.7. The denotation of a proof π of type $\mathbf{bint}_\alpha$ is a polynomial function

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

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. For example,

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

The idea of the second-order RNN is to view the input and output symbols as linear operators on a vector space $\mathscr{K}$ (which, following the idea of the multiplicative RNN, we take as possibly different from the hidden state space $\mathscr{H}$). Linear logic gives us a source language in which to create programs, whose denotations are differentiable maps taking linear operators as input and providing linear operators as output. This map from input to output may not itself be linear, as the above examples show. It is these differentiable shadows of symbolic transformations in linear logic that we will couple to the RNN.

Let us conclude with an example involving more complicated types:

Example 2.8. There is a program

$$\underline{\mathit{subst}} : \mathbf{bint}_\alpha \multimap \Big(!\mathbf{bint}_\alpha \multimap \big(!\mathbf{bint}_\alpha \multimap \mathbf{bint}_\alpha\big)\Big)$$

which given three input sequences $\underline{S}, \underline{T}, \underline{U}$ uses the pattern of the sequence S to intertwine copies of T, U. A program, say $\underline{T}$, of type $\mathbf{bint}_\alpha$ may be "promoted" to a program of type $!\mathbf{bint}_\alpha$, denoted $[\underline{T}]$. Then, for example,

$$\underline{\mathit{subst}}(\underline{010}, [\underline{T}], [\underline{U}]) = \underline{T \cdot U \cdot T} : \mathbf{bint}_\alpha$$

where $\cdot$ means concatenation.

2.3 The linear logic RNN

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 [13, §]. 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.

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 at each time step diagonal matrices $\{p_\lambda^{(t)}\}_{\lambda\in\Lambda}$ and $\{q_\pi^{(t)}\}_{\pi\in\Pi}$ that determine which programs are used.

Definition 2.9. 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, I, J and an evolution equation:

$$\begin{aligned} p_\lambda^{(t+1)} &= \sigma\big(P_\lambda h^{(t)} + C_\lambda\big)\,, \\ q_\pi^{(t+1)} &= \sigma\big(Q_\pi h^{(t)} + D_\pi\big)\,, \\ h^{(t+1)} &= \sigma\Big(Z + Hh^{(t)} + Ux^{(t+1)} + B\Big) \end{aligned}$$

where

$$Z = \sum_{\lambda\in\Lambda} I p_\lambda^{(t+1)} [\![\lambda]\!](Vx^{(t+1)})\big(Jh^{(t)}\big) + \sum_{\pi\in\Pi} I q_\pi^{(t+1)} [\![\pi]\!](Vx^{(t)}, Vx^{(t+1)})\big(Jh^{(t)}\big)\,.$$

The weight matrices V, I, J are interpreted as linear maps as in (2.6). The operators $p_\lambda^{(t)}$ and $q_\pi^{(t)}$ are diagonal matrices in $\mathrm{End}_{\mathbb{R}}(\mathscr{K})$. 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.

Example 2.10. If $\Lambda = \{\underline{1}, \underline{2}\}$ and $\Pi = \{\underline{010}\}$ then

$$\begin{aligned} Z = {} & I\big(p_1^{(t)} V(x^{(t)})(Jh^{(t)})\big) + I\big(p_2^{(t)} V(x^{(t)})^2(Jh^{(t)})\big) \\ & + I\Big(q_{010}^{(t)}\big\{V(x^{(t)}) \circ V(x^{(t+1)}) \circ V(x^{(t)})\big\}(Jh^{(t)})\Big)\,. \end{aligned}$$

The first summand here is the same as (2.7), which shows that the linear logic RNN with $\Lambda = \{\underline{1}\}, \Pi = \emptyset$ and the weight $p_1^{(t)}$ fixed to the identity, is the multiplicative RNN of [7].

Remark 2.11. To avoid too many weights and thus slow training, we follow the multiplicative RNN in assuming that V maps inputs to diagonal matrices. But the whole point of adding the ability to call programs from Π is that binary integers intertwine $V(x^{(t)})$ and $V(x^{(t+1)})$ in some useful way, which is compromised if these two matrices commute with one another.

So in practice we have two auxiliary spaces $\mathscr{K}_1, \mathscr{K}_2$ and two sets of weights V_i, I_i, J_i. The index $i = 1$ corresponds to Λ, and $V_1 : \mathscr{I} \longrightarrow \mathrm{End}_{\mathbb{R}}(\mathscr{K}_1)$ factors through the subspace of matrices diagonal in a chosen basis. The map V_2 is not restricted in this way, but to compensate we take $\dim(\mathscr{K}_2) < \dim(\mathscr{K}_1)$.

Remark 2.12. A more powerful system would substitute a neural theorem prover along the lines of [22, 24] in place of the library of functional programs Π, Λ. 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 [23].

Remark 2.13. Although intuitionistic linear logic is a weak system in terms of expressive strength the difficulty of training the RNN with polynomial functions of high degree makes it infeasible to make use of even this much strength. It would, however, be very interesting to define differentiable semantics for second-order logic and try to use this to improve the generalisation ability of neural networks.

2.4 Differential linear logic

In functional programming languages like differential λ-calculus [30] and differential linear logic [25] 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 not as exotic as it might sound: it is a generalisation of automatic differentiation [29] which is widely used to compute derivatives of complicated real-valued functions, for example in the backpropagation algorithms of machine learning packages like TensorFlow [27, §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 [31]. 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.

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 $h^{(t-1)}$.

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

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
```

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 $h_0^{(1)}, h_0^{(2)}, \ldots$ 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.

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).

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

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

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

[16] J.-Y. Girard, *Linear Logic*, Theoretical Computer Science **50** (1987), 1–102.

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

[18] E. Grefenstette, et al, *Learning to transduce with unbounded memory*, Advances in Neural Information Processing Systems, 2015.

[19] J. Weston, C. Sumit and B. Antoine, *Memory networks*, preprint (2014) [arXiv:1410.3916].

[20] W. Zaremba, et al., *Learning Simple Algorithms from Examples*, preprint (2015) [arXiv:1511.07275].

[21] 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.

[22] A. A. Alemi, F. Chollet, G. Irving, C. Szegedy and J. Urban, *DeepMath-Deep Sequence Models for Premise Selection*, arXiv preprint arXiv:1606.04442.

[23] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin and D. Tarlow, *DeepCoder: Learning to Write Programs*, arXiv preprint arXiv:1611.01989.

[24] T. Rocktäschel and S. Riedel, *Learning Knowledge Base Inference with Neural Theorem Provers*, In NAACL Workshop on Automated Knowledge Base Construction (AKBC) 2016.

[25] T. Ehrhard, *An introduction to Differential Linear Logic: proof-nets, models and antiderivatives*, [arXiv:1606.01642] (2016).

[26] J. Clift and D. Murfet, *The Sweedler semantics is a natural model of differential linear logic*, preprint.

[27] Abadi, Martn, et al. *TensorFlow: A system for large-scale machine learning* arXiv preprint arXiv:1605.08695 (2016).

[28] Abadi, Martn, et al. *Tensorflow: Large-scale machine learning on heterogeneous distributed systems*, arXiv preprint arXiv:1603.04467 (2016).

[29] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*, Siam (2008).

[30] T. Ehrhard and L. Regnier, *The differential λ-calculus*, Theoretical Computer Science 309, pp. 1–41, (2003).

[31] 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].