Training Neural Networks to Do Logic, with Logic

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
paul.tarau@unt.edu

Abstract. We overview combinatorial generation algorithms, with focus on lambda terms and related type inference algorithms, all elegantly expressible in a logic programming language that supports backtracking and unification. With help from these tools, we introduce methods to train neural networks as theorem provers. Our combinatorial generation algorithms provide pairs of lambda terms together with their inferred types. We make use of the Curry-Howard isomorphism between lambda terms and formulas in linear and intuitionistic logic corresponding to their types to train our neural networks on large, combinatorially generated datasets mapping formulas to their proof terms.

Keywords: logic programming tools for theorem proving, intuitionistic and linear logic, Curry-Howard isomorphism, neural theorem proving, neuro-symbolic computing

Renewed interest in neuro-symbolic computing [1, 2] and specifically experiments with training neural networks on theorem proving tasks [3] open the doors for applications of logic programming-based techniques and tools both as explainable AI interfaces and symbiotic logic and neural systems, combining deductive strengths with pattern retrieval via machine learning.

SLD-resolution based logic programming languages (and in particular Prolog) provide a unified framework for implementing combinatorial generation algorithms, type-inference as well as search-intensive theorem provers.

Generating formulas of a given size in Prolog is quite easy as exemplified by the following code snippet, generating all implicational formulas of size $N$, counted by the well-known Catalan numbers.

```prolog
gen_tree(N,Tree,Leaves):-gen_tree(Tree,N,0,Leaves,[]).

gen_tree(V,N,N,[V|Vs],Vs).

gen_tree((A -> B),SN1,N3,Vs1,Vs3):-pred(SN1,N1),
gen_tree(A,N1,N2,Vs1,Vs2),
gen_tree(B,N2,N3,Vs2,Vs3).

pred(SN,N):-succ(N,SN).
```

The counts of generated trees match entry A000108 in [4], representing the Catalan numbers [5], binary trees with $N$ internal nodes.

All possible canonical labelings of variables can be derived from a set-partitioning algorithm, counted by the Bell numbers for the $N + 1$ leaves of the trees with $N$ internal nodes.
Similar generation algorithms can be built for several families of lambda terms (see [6]).

A theorem prover for intuitionistic propositional logic is derived directly from the sequent calculus formulas of Gentzen’s LJ calculus, modified by Roy Dyckhoff [7] with a rewriting rule for nested implications that avoids the need for loop checking.

In its simplest form, the Curry-Howard isomorphism [8] connects the implicational fragment of propositional intuitionistic logic with types in the simply typed lambda calculus. A low polynomial type inference algorithm associates a type (when it exists) to a lambda term. Harder (PSPACE-complete, see [9]) algorithms associate inhabitants to a given type expression with the resulting lambda term (typically in normal form) serving as a witness for the existence of a proof for the corresponding tautology.

To train neural networks as theorem provers via the Curry-Howard isomorphism we need to efficiently generate all theorems up to a given size in the implicational fragment of propositional intuitionistic or linear logic, with help from their lambda term counterparts.

We will apply this mechanism to formulas inferred as types of simply typed lambda terms. After designing a compact encoding of the formula and proof term pairs we will discuss our experiments with training a seq2seq LSTM recurrent network [10] which will perform, in inference mode, with 92% accuracy as a theorem prover for the implicational fragment of IPC on unseen formulas from a test dataset.

Linear Logic [11], as a resource-control mechanism, constrains the use of formulas available as premises in a proof. We will reflect this constraint when generating lambda terms seen as proof terms of our formulas, on the other side of the Curry-Howard isomorphism. Linear Logic’s proof mechanism is known to be Turing-complete even in the propositional case. Thus, we will focus on generating all theorems of a given size for a restricted set of formulas, the Implicational fragment of Propositional Intuitionistic Linear Logic (IPILL), corresponding to principal types of lambda terms in normal form.

We start by filtering for linearity the proof terms associated by a Prolog-based theorem prover for Implicational Intuitionistic Logic. This works, but using for each formula a PSPACE-complete algorithm limits it to very small formulas. We take a few back and forth walks over the bridge between proof terms and theorems, provided by the Curry-Howard isomorphism, and derive step-by-step an efficient algorithm requiring a low polynomial effort per generated theorem. The resulting Prolog program runs in $O(N)$ space for terms of size $N$ and generates in a few hours $7,566,084,686$ theorems in IPILL, together with their proof terms.

The figure below shows a lambda term normal form and its corresponding linear type (where we label lambda nodes with l, application nodes with a, linear implication nodes with -o and lambda and type variables with uppercase letters). Note the symmetries present in this “Goldilocks” special case between formulas and their proof terms, largely responsible for our interest in them.
As terms and clauses in logic programming languages have a tree (or equivalently, a directed acyclic graph) representation, easy to linearize to a canonical representation for both the theorems and their proofs.

Training the Neural Networks as Theorem Provers via the Curry-Howard Isomorphism proceeds as follows. Formulas/types and proofs/lambda terms are both trees, thus we can represent them as prefix strings. Note that for IPILL we can even find a size definition to give the same size on both sides:

- for lambda terms: leaves=0, lambda nodes=1, applications=1
- for –o formulas: leaves=0, lollipops = 1

We generate prefix encodings of formulas with lollipop=0, application=0, lambda=1, variables as uppercase letters, “::” as separator between formulas and proof terms.

Example 1 Provable formulas with their proof terms (for IPILL)

<table>
<thead>
<tr>
<th>Formula</th>
<th>Proof Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>0AA:1AA</td>
<td></td>
</tr>
<tr>
<td>0A00ABB:1A1BOBA</td>
<td></td>
</tr>
<tr>
<td>00AB0AB:1A1BOAB</td>
<td></td>
</tr>
<tr>
<td>0A00ABO0BCC:1A1B1OC0C0BA</td>
<td></td>
</tr>
<tr>
<td>00000AAB00C0B00C0D00C0E0F0FF:1A00A1B1C1D00C0D0B1EE1FF</td>
<td></td>
</tr>
</tbody>
</table>

Example 2 Provable formulas with their proof terms and “?” if proof failed

<table>
<thead>
<tr>
<th>Formula</th>
<th>Proof Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>0A0B0000A0C0B0DE0C0D0E0F00000DABEF</td>
<td></td>
</tr>
<tr>
<td>0A0B0000A0C0B0DE0C0D0F00000DABEF</td>
<td></td>
</tr>
<tr>
<td>0A0B0000A0C0B0DE0C0F00000DABEF</td>
<td></td>
</tr>
<tr>
<td>0A0B0000A0C0B0DE0C0F00000DABEF</td>
<td></td>
</tr>
</tbody>
</table>

Similar formulas cover implicational intuitionistic propositional logic IIPC, also with normal forms in prefix notation.

Recurrent Neural Networks (RNN) keep track of dependencies within sequences. They work by using feedback from values at time t that is fed into computations at time t+1.
Among them, a long short-term memory (LSTM) network is a recurrent neural network architecture that can not only process single data points (such as images), but also entire sequences of data (such as text, speech or video). LSTM neural networks have feedback connections, that help them avoid vanishing or exploding gradient problems by also feeding unchanged values to the next layer. Our neural networks will learn the mapping between theorems and their proof-terms on arbitrarily large datasets generated by inferring the type of simply typed lambda terms in normal form, corresponding to formulas via the Curry-Howard correspondence. In particular, theorems in the linear fragment of IIPC will be obtained as types of linear lambda terms.

With trees as prefix string we will use “seq2seq” recurrent neural networks, in particular LSTM (long short term memory) networks that are good to handle long distance dependencies in the prefix terms, in a similar way to their original use for Natural Language processing applications. We plan to also experiment in the future with newer variants, possibly more in interesting: tree2tree, dag2dag and several types of graph neural networks (e.g., convolutional, attention, spectral, torch geometric).

We have evaluated the performance of our neural networks working as theorem provers and our seq2seq LSTM recurrent neural network trained on encodings of theorems and their proof-terms performs unusually well. Besides the usual accuracy/loss curves, we have also tested them on unseen formulas, with 100/100 success on IPILL and 92/100 success on IIPC formulas.

We explain these unusually good results for IPILL in terms of a size-preserving bijection between linear lambda terms in normal form and their principal types. The bijection, first proven in [12] and given a geometric interpretation in [13] is based on a reversible transformation of oriented edges in the tree describing a linear lambda term in normal form, into corresponding oriented edges in the tree describing the linear implicational formula, acting as its principal type. Thus, by generating all typable linear lambda terms in normal form size $N$, we obtain, for free, a generator for all corresponding IPILL theorems of size $N$.

The dataset, containing generated theorems of several sizes and their proof-terms in postfix form, is available at http://www.cse.unt.edu/~tarau/datasets/ltltaut/ and can be used for correctness, performance and scalability testing for linear logic theorem provers. Our experiments with training Recurrent Neural Networks using our
Fig. 2. Loss curve of the LSTM seq2seq neural network on our formula/proof term dataset for IPILL.

Fig. 3. Accuracy for IPILL + unprovable formulas.

Fig. 4. Loss for IPILL + unprovable formulas.

Fig. 5. Accuracy for IIPC.
implicational linear logic theorem dataset are available at: https://github.com/ptarau/neuralgs.

Conclusion

There are basically two ways neural networks have been used to help with theorem proving. Theorem provers are computation-intensive search algorithms often Turing-complete (e.g., propositional linear logic, first order predicate calculus) or PSPACE-complete (e.g., IPC). Thus, a first approach is to help with fine-tuning the search, by selecting the right choices at choice points. Another way to integrate neural networks in symbolic computations has been via an interface to solve low-level “perception”-intensive tasks (e.g., working on learnable ground facts labeled with probabilities).

Our experiments hint to a third way that is more radical but also much simpler: we replace the symbolic theorem prover with a neural network trained on a large enough training dataset.

References