Training Neural Networks to Do Logic, with Logic

Paul Tarau

joint work with Valeria de Paiva

February 22, 2021

Overview

- THE PROBLEM:
  - can we train neural networks to work as close-to-perfect theorem provers on an interesting logic?
- OUR SOLUTION:
  - we focus on a simple enough, but interesting logic: Implicational Propositional Intuitionistic Linear Logic (**IPILL** from now on)
  - we need to derive an efficient algorithm requiring a low polynomial effort per generated theorem and its proof term
  - $\Rightarrow$ we rely on the Curry-Howard isomorphism $\Rightarrow$ we can focus on generating simply typed linear lambda terms in normal form
- THE OUTCOMES:
  - an implicational intuitionistic logic prover specialized to **IPILL** formulas
  - a dataset for training neural networks
  - very high success rate with **seq2seq LSTM** neural networks
- an open problem: can these techniques extend to harder, syntactically and semantically richer logics?

The Tools Used

- we have designed a combinatorial generation framework for several formula languages:
    - exhaustive generators for terms up to a given size
    - random term generators
    - families of lambda terms (including linear lambda terms)
    - type inference algorithms
    - generators of lambda terms constrained by typability
    - theorem provers for IPC
- we have chosen Prolog as our meta-language, because:
    - it reduces the semantic gap (derived from essentially the same formalisms as those we are covering)
    - has the right language constructs for a concise and efficient declarative implementation
- the Prolog code is available at: **`https://github.com/ptarau/TypesAndProofs`**.

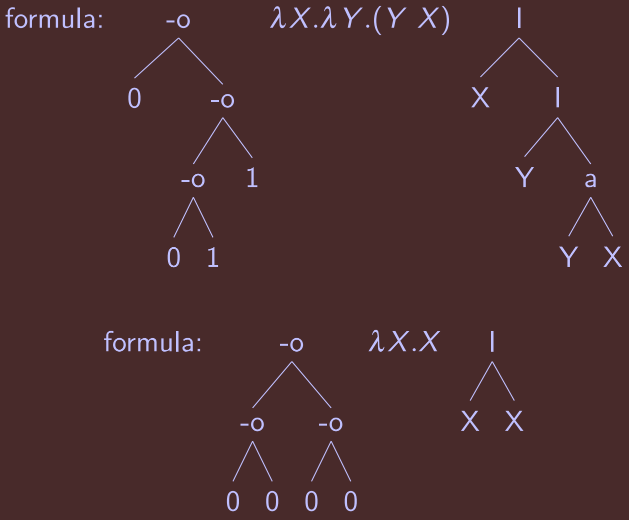
The Implicational Fragment of Propositional Intuitionistic Linear Logic (IPILL)

- while propositional intuitionistic linear logic is already Turing complete, its *implicational fragment* is decidable
- moreover, via the Curry-Howard isomorphism, we can design (polynomial) algorithms for generating its theorems and their proofs
- dual uses of theorems and their proofs (expressed als linear lambda terms)
    - as *test sets*, combining tautologies and their proof terms helps with testing correctness and scalability of linear logic theorem provers
    - as *datasets*, they can be used for training deep learning networks focusing on *neuro-symbolic* computations

The Curry Howard Isomorphism

- a correspondence between *computations* and *proofs* : the *Curry-Howard isomorphism*
- in its simplest form, it connects the *implicational fragment of propositional intuitionistic logic* **IIPC** 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) 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 in implicational propositional intuitionistic logic
- ⇒ can we use combinatorial generation of lambda terms + type inference (easy) to "solve" some type inhabitation problems (hard)?

Formulas depicted as trees, together with their proof terms



formula: $\lambda X.\lambda Y.(Y\ X)$

formula: $\lambda X.X$

Deriving the formula generators (see ICLP'20 paper)

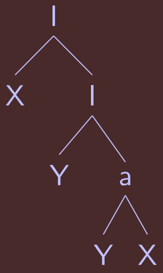
1. **IPILL** formulas (fairly simple Prolog code), built as:
   - binary trees of size N, counted by Catalan numbers $Catalan(N)$
   - labeled with variables derived from set partitions counted by $Bell(N+1)$ (see **A289679** in OEIS)
2. linear lambda terms (proof terms for the **IPILL** formulas)
   - linear skeleton Motzkin trees (binary-unary trees with constraints enforcing one-to-one mapping from variables to their lambda binders)
3. *closed* linear lambda terms
4. closed linear lambda terms in normal form
5. after a chain of refinements, we derive a compact and efficient generator for *pairs of Linear Lambda Terms in Normal Form* and their types (which always exist as they are all typable!) see next slide!
6. it generates in a few hours **7,566,084,686** terms together with their corresponding types, seen as theorems in **IPILL** via the Curry-Howard isomorphism (**A062980** sequence in OEIS)

The Linear Lambda Term in Typed Normal Form Generator

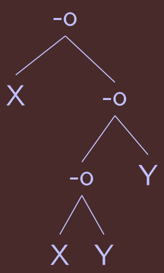
```
linear_typed_normal_form(N,E,T):-succ(N,N1),
  linear_typed_normal_form(E,T,N,0,N1,0,[]).

linear_typed_normal_form(l(X,E),(S '-o' T),A1,A2,L1,L3,Vs):-
  pred(L1,L2), % defined as L1>0,L2 is L1-1
  linear_typed_normal_form(E,T,A1,A2,L2,L3,[V:S|Vs]),
  check_binding(V,X).
linear_typed_normal_form(E,T,A1,A2,L1,L3,Vs):-
  linear_neutral_term(E,T,A1,A2,L1,L3,Vs).

linear_neutral_term(X,T,A,A,L,L,Vs):-
  member(V:TT,Vs),bind_once(V,X),T=TT.
linear_neutral_term(a(E,F),T,A1,A4,L1,L3,Vs):-pred(A1,A2),
  linear_neutral_term(E,(S '-o' T),A2,A3,L1,L2,Vs),
  linear_typed_normal_form(F,S,A3,A4,L2,L3,Vs).

bind_once(V,X):-var(V),V=v(X).
check_binding(V,X):-nonvar(V),V=v(X).
```

A Normal Form and its Corresponding Linear Type (I).

term: $\lambda X.\lambda Y.(Y\ X)$



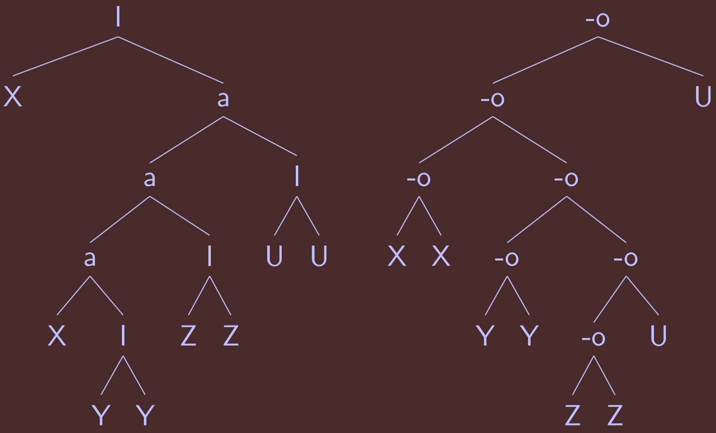
its linear type:



Note that all linear lambda terms are typable!

A Normal Form and its Corresponding Linear Type (II).

$\lambda X.(((X\ \lambda Y.Y)\ \lambda Z.Z)\ \lambda U.U)$



Note the symmetries between linear terms and their types!

An Eureka Moment

- it looks like we see some interesting symmetries in the pictures!
  - there are exactly two occurrences of each variable both in the theorems and their proof terms of which they are the principal types
  - theorems and their proof terms *have the same size*, counted as number of internal nodes
- *thus, we can solve the problem of generating all* **IPILL** *tautologies size N*

  IF

  the predicate `linear_typed_normal_form` *implements a generator of their proof-terms of size N*

Theorems for Free: the Size Preserving Bijection

- the GOOD NEWS: there's a *size-preserving bijection between linear lambda terms in normal form and their principal types*!
- a proof follows immediately from a paper by Noam Zeilberger who attributes this observation to Grigori Mints
- ⇒ we have obtained a generator for all theorems of implicational linear intuitionistic propositional logic of a given size, as measured by the number of lollipops, without having to prove theorems!
- this is a "Goldilocks" situation that points out the very special case that implicational formulas have in linear logic and equivalently, linear types have in type theory!

The Datasets

- the dataset containing generated theorems and their proof-terms in prefix form (as well as their LaTeX tree representations marked as Prolog "%" comments) is available at `http://www.cse.unt.edu/~tarau/datasets/lltaut/`
- it can be used for correctness, performance and scalability testing of linear logic theorem provers
- the `<formula, proof-term>` pairs in the dataset are usable to test deep-learning systems on theorem proving tasks
- also, formulas with non-theorems added for **IPILL**

Examples of Data records

prefix encoding: lollipop=0, application=0, lambda=1, variables as uppercase letters, ":" as separator between formulas and proof terms

- Provable formulas with their proof terms (for **IPILL**)

```
0AA:1AA
0A00ABB:1A1B0BA
00AB0AB:1A1B0AB
0A00AB00BCC:1A1B1C0C0BA
00000AAB00C0BD0CD00EEFF:1A00A1B1C1D00CD0B1EE1FF
```

- Provable formulas with their proof terms and "**?**" if proof failed

```
0A0B0000A0C0B0DE0C0DEFF:1A1B1C0C1D1E1F0000DAEBF
0A0B0000A0C0B0DE0C0DFGH:?
0A0B0000A0B0C0DE0D0CEFF:1A1B1C0C1D1E1F0000DABFE
0A0B0000A0B0C0DE0D0CFGG:?
```

- similar formulas for **IPC**, also on normal forms in prefix form

How can Neural Networks help with Theorem Proving?

- more generally, we search for good frameworks for neuro-symbolic computing
- theorem provers are computation-intensive search algorithms
- Turing-complete (e.g., PLL, FOL), PSPACE-complete (e.g., IPC)
- there are two ways neural networks can help:
  - fine-tuning the search, by helping with the right choice at choice points
  - used via an interface to solve low-level “perception”-intensive tasks
    - e.g., working on learnable ground facts labeled with probabilities – DeepProbLog
    - also, via an interface to a ground term Prolog database: (see `https://github.com/ptarau/pypro`)
- is there a third way: can they simply replace the symbolic theorem prover given a large enough training dataset?

Machine Learning (ML) with Deep Neural Networks (NNs)

- the key ML concepts to watch for:
  - “honesty”: split the dataset into: training, validation and (independent) test sets
  - things to avoid:
    - overfitting (works on training, fails on validation and testing data)
    - unlikely to work well on random (high Kolmogorov complexity) data
- the key NN general concepts to watch for:
  - NNs are *trainable universal approximators* for a given function
  - $L_{t+1} = \sigma(A * L_t + b)$ where L_t is a layer at step t, A is a matrix containing trainable parameters, b is a bias vector and σ is a non-linear function (logistic sigmoid, tanh, RELU(x)=max(0,x), etc.)
  - differentiable functions, gradients computed on backpropagation
  - an intuition behind why deep NNs are needed: each layer abstracts away statistically relevant patterns that are fed to the next layer
  - often, to ensure generalization, information is deliberately lost

Training the Neural Networks as Theorem Provers via the Curry-Howard Isomorphism

- formulas/types and proofs/lambda terms are both trees
- $\Rightarrow$ we can represent them as prefix strings
- $\Rightarrow$ 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 $\multimap$ formulas: leaves=0, lollipops = 1
- what type of neural networks to use?
    - with trees as prefix string: $\Rightarrow$ "seq2seq" recurrent NNs
    - LSTM (long short term memory) NNs : good to handle long distance dependencies in the prefix forms

seq2seq Neural Networks

- sequence as input, train to guess sequence as output
- used originally for translation of natural languages, with training on large parallel corpora
- notable variants: *transformers*, trained to predict masked words in a sentence as well as predict next sentence in a text
- *unsupervised* - just feeding them very large text data
- examples: BERT, GPT-3 - impressive performance on several NLP tasks (e.g., GPT-3 generating fake news)
- newer variants, possibly more in interesting: **tree2tree**, **dag2dag** and several types of **graph neural networks** (e.g., convolutional, attention, spectral, torch geometric)

LSTM seq2seq Neural Networks

- recurrent neural networks keep track of dependencies within sequences
- feedback from values at time t is fed into computations at time $t+1$
- long short-term memory (LSTM) is a recurrent neural network (RNN) architecture
- it can not only process single data points (such as images), but also entire sequences of data (such as text, speech or video)
- LSTM NNs have feedback connections $\Rightarrow$ LSTM avoids vanishing or exploding gradient problems by also feeding *unchanged* values to the next layer

Evaluating the Performance of our Neural Networks as Theorem Provers

- in fact, our `seq2seq` LSTM recurrent neural network trained on encodings of theorems and their proof-terms performs unusually well
- the experiments with training the neural networks using the IPILL and IIPC theorem dataset are available at: `https://github.com/ptarau/neuralgs`
- the $< \mathit{formula},\ \ \mathit{proof\ \ term} >$ generators are available at: `https://github.com/ptarau/TypesAndProofs`
- the generated datasets are available at: `http://www.cse.unt.edu/~tarau/datasets/`

Accuracy of the LSTM seq2seq neural network on our formula/proof term dataset for **IPILL**

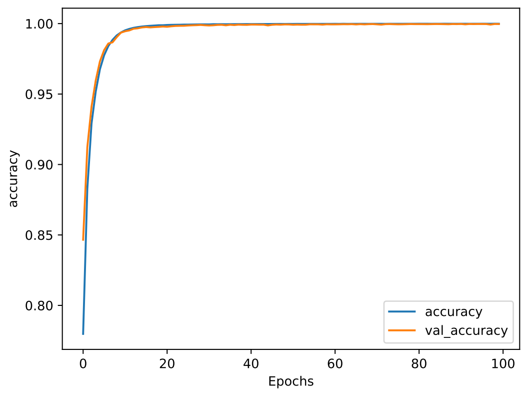


Figure: Accuracy curve for 100 epochs

Loss curve of the LSTM seq2seq neural network on our formula/proof term dataset for **IPILL**

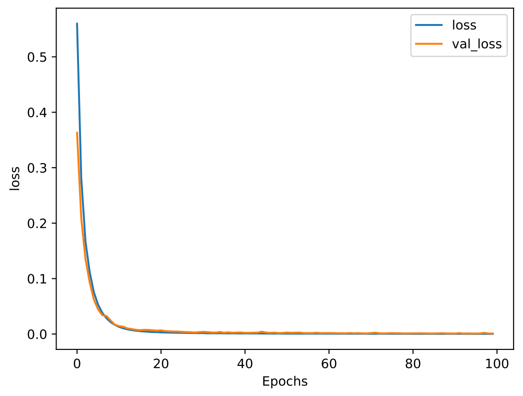


Figure: Loss curve for 100 epochs

Accuracy for **IPILL** + unprovable formulas

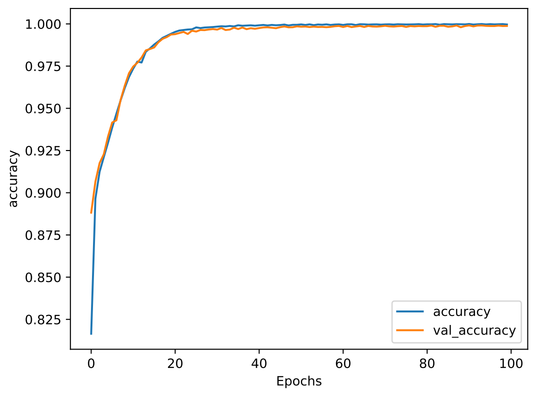


Figure: Accuracy curve for 100 epochs

Loss for **IPILL** + unprovable formulas

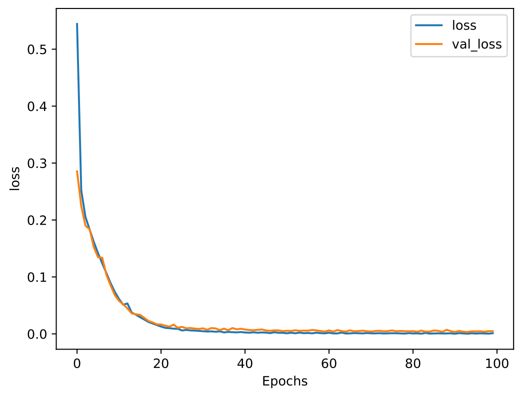


Figure: Loss curve for 100 epochs

A harder Logic: Implicational Intuitionist Propositional Logic

Can we train Neural Network as Provers for a PSPACE-complete Logic?

The **LJT/G4ip** calculus (implicational fragment)

Roy Dyckhoff's rules for the **G4ip** (originally called the **LJT**)

$$LJT_1 : \quad \frac{}{A,\Gamma \vdash A}$$

$$LJT_2 : \quad \frac{A,\Gamma \vdash B}{\Gamma \vdash A{\rightarrow}B}$$

$$LJT_3 : \quad \frac{B,A,\Gamma \vdash G}{A{\rightarrow}B,A,\Gamma \vdash G}$$

$$LJT_4 : \quad \frac{D{\rightarrow}B,\Gamma \vdash C{\rightarrow}D \qquad B,\Gamma \vdash G}{(C{\rightarrow}D){\rightarrow}B,\Gamma \vdash G}$$

$$LJT_5 : \quad \frac{}{\mathit{false},\Gamma \vdash G}$$

the last rule supports intuitionistic negation

A Lightweight Theorem Prover for Full Intuitionistic Propositional Logic

the LJT/G4ip sequent calculus for the full IPC + rules for “<->”:

```
ljfa(T):-  ljfa(T,[]).

ljfa(A,Vs):-memberchk(A,Vs),!.
ljfa(_,Vs):-memberchk(false,Vs),!.
ljfa(A<->B,Vs):-!,ljfa(B,[A|Vs]),ljfa(A,[B|Vs]).
ljfa((A->B),Vs):-!,ljfa(B,[A|Vs]).
ljfa(A & B,Vs):-!,ljfa(A,Vs),ljfa(B,Vs).
ljfa(G,Vs1):- % atomic or disj or false
  select(Red,Vs1,Vs2),
  ljfa_reduce(Red,G,Vs2,Vs3),
  !,
  ljfa(G,Vs3).
ljfa(A v B, Vs):-(ljfa(A,Vs);ljfa(B,Vs)),!.
```

continued

```
ljfa_reduce((A->B),_,Vs1,Vs2):-!,ljfa_imp(A,B,Vs1,Vs2).
ljfa_reduce((A & B),_,Vs,[A,B|Vs]):-!.
ljfa_reduce((A<->B),_,Vs,[(A->B),(B->A)|Vs]):-!.
ljfa_reduce((A v B),G,Vs,[B|Vs]):-ljfa(G,[A|Vs]).

ljfa_imp((C->D),B,Vs,[B|Vs]):-!,ljfa((C->D),[(D->B)|Vs]).
ljfa_imp((C & D),B,Vs,[(C->(D->B))|Vs]):-!.
ljfa_imp((C v D),B,Vs,[(C->B),(D->B)|Vs]):-!.
ljfa_imp((C<->D),B,Vs,[((C->D)->((D->C)->B))|Vs]):-!.
ljfa_imp(A,B,Vs,[B|Vs]):-memberchk(A,Vs).
```

- Being derived from a sound and complete calculus, our prover is sound and complete. Also, it is safe from stack and heap overflows.
- We can use it as an oracle for validating the output of a neural network on larger, unknown formulas.
- However, as we can generate such formulas up to size N directly, and then infer their types, we can validate the results on the dataset itself, split into training, validation and test sets.

Accuracy for **IIPC**

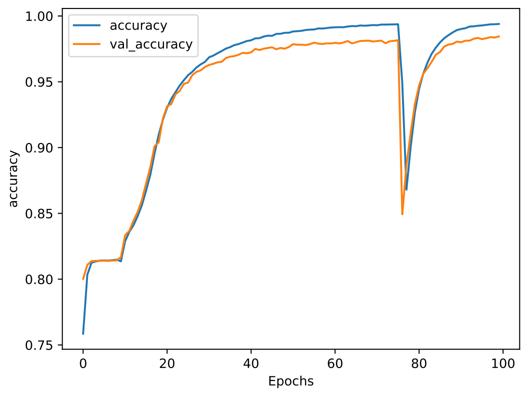


Figure: Accuracy curve for 100 epochs

Loss for **IIPC**

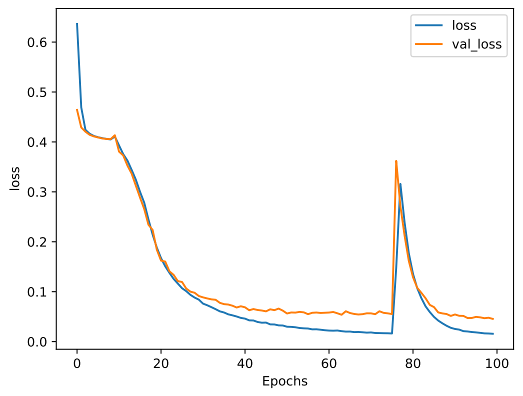


Figure: Loss curve for 100 epochs

Conclusions

- we have used a Logic Programming Language (Prolog) to derive a generator for all **IPILL** and **IIPC** theorems of a given size, without needing a theorem prover by combining a generator for their proof terms and a type inference algorithm
- we have sketched their use as a dataset for training neural networks, turning them into reliable theorem provers, for the harder inverse problem: given a formula in **IPILL**, or **IIPC**, find a proof term for it!
- open problems, future work:
  - can this be extended to full fragments of IPC or LL?
  - would the same success rate apply to large, random generated formulas?
  - how would the NNs perform on larger, human-made formulas?