Machine Learning Coalgebraic Proofs

Ekaterina Komendantskaya[1]

Department of Computing, University of Dundee, UK *

Abstract. This paper presents a method to machine learn formal proofs using neural networks. The method exploits coalgebraic approach to proofs. The success of the method is demonstrated on three applications allowing to distinguish well-formed proofs from ill-formed proofs, identify families of proofs and even families of potentially provable goals.
Key words: Logic Programming, Coalgebra, Coinductive Proofs, Statistical Machine Learning, Neural Networks

1 Introduction

Mathematical proofs can be developed in a formal language and within well-defined logical (deductive) theory; this reasoning can even be done by automated theorem provers. However, in practice, some steps in formal reasoning may have statistical or inductive nature, cf. [1, 2, 4]. This paper proposes a method of using statistical machine learning in analysis and implementation of formal proofs.

Higher-order interactive theorem provers (e.g. HOL or Coq) have been successfully developed into complicated environments for mechanised proofs. Whether these provers are applied to big industrial tasks in software verification, or to formalisation of mathematical theories, a programmer may have to tackle thousands of lemmas and theorems of variable sizes and complexities. A proof in such languages is constructed by combining a finite number of tactics. Some proofs may yield the same pattern of tactics, and can be fully automated, and others may require programmer's intervention. Discovery of such proof tactics may be one area of application of machine learning and pattern recognition.

Another feature of theorem proving is that unsuccessful attempts of proofs, although discarded when the correct proof is found, play an important role in proof discovery. This kind of "negative" information finds no place in mathematical textbooks or libraries of automated proofs. Conveniently, analysis of both positive and negative examples is inherent in statistical machine learning [3].

However, applying statistical machine-learning methods to generalise or classify data coming from proof theory is a challenging task for several reasons. To mention a few, formulae written in formal language have precise, rather than statistical nature. For example, `list(nil)` may be a well-formed term, while `list(nol)` - not; although they may have similar patterns recognisable by machine learning methods. Another problem is that many algorithms applied in

* The work was supported by the Engineering and Physical Sciences Research Council, UK; Postdoctoral Fellow research grant EP/F044046/2.

proof theory are sequential, such as e.g. first-order unification, while most statistical learning methods perform parallel processing of vectors of data.

As a solution to the outlined problems, the paper shows that *coalgebraic* approach to proofs may provide the right area for applications of machine learning methods. Firstly, coalgebraic computations are often concurrent, and this may be the key to obtaining adequate vector representations of the problems. Secondly, they are based on the idea of repeating patterns of potentially infinite computations. These patterns mimic the internal term structure, which may be detected by methods of statistical pattern recognition, [3].

This paper presents a method to recognise coinductive proofs using machine learning. Section 2 introduces the coalgebraic approach to proofs in first-order logic programming. Section 3 formulates a suitable representation of the coinductive proofs trees. Section 4 uses this representation to train neural networks for three different tasks. Section 5 concludes.

2 Coinductive proofs and proof-trees

We assume that the reader is familiar with the basic techniques of Logic Programming [8]. We start with an example of a logic program.

Example 1. Let ListNat denote the logic program consisting of clauses

1. `nat(0)` $\leftarrow$
2. `nat(s(x))` $\leftarrow$ `nat(x)`
3. `list(nil)` $\leftarrow$
4. `list(cons x y)` $\leftarrow$ `nat(x), list(y)`

The algorithm of SLD-resolution [8] is a sequential proof-search algorithm.

Example 2. For a goal G_0 = `list(cons(x, cons(y, x)))`, SLD-resolution produces a sequence of proof steps: G_1 = `nat(x),list(cons(y, x))`, G_2 = `list(cons(y,0))`, G_3 = `nat(y),list(0)`, G_4 = `list(0)`, G_5 = `fail`. If we consider applications of each of the clauses 1-4 as tactics, and also tactics 5 and 6 for "fail" and "succeed", then the proof could be represented as 4,1,4,1,5. It is a well-formed proof, although the derivation fails; 4,1,4,1,6 - would be ill-formed.

We briefly recall the definition of the coinductive derivation trees from [7].

Definition 1. *Let P be a logic program and $G =\leftarrow A$ be an atomic goal. The* coinductive derivation tree *for A is a possibly infinite tree T satisfying the following properties.*

- *A is the root of T.*
- *Each node in T is either an and-node or an or-node: Each or-node is given by $\bullet$. Each and-node is an atom.*

– *For every and-node A' occurring in T, there exist exactly $m > 0$ distinct clauses $C_1, \ldots, C_m$ in P (a clause C_i has the form $B_i \leftarrow B_1^i, \ldots, B_{n_i}^i$, for some n_i), such that $A' = B_1\theta_1 = \ldots = B_m\theta_m$, for some substitutions $\theta_1, \ldots, \theta_m$, then A' has exactly m children given by or-nodes, such that, for every $i \in m$, the ith or-node has n children given by and-nodes $B_1^i\theta_i, \ldots, B_{n_i}^i\theta_i$.*

Note that, comparing this with the SLD-resolution algorithm, the definition of coinductive derivation tree introduces *concurrent*, and not sequential derivations. It restricts unification to the case of *term matching*, i.e., the substitution θ unifying atoms A_1 and A_2 is applied only to one atom, e.g. $A_1 = A_2\theta$, whereas traditionally mgus satisfy $A_1\theta = A_2\theta$. The term-matching algorithm is parallelisable, in contrast to the unification algorithm, which is P-complete.

Coinductive derivations resemble *tree rewriting*. The notion of a successful proof is captured by the definition of *success subtrees* [7], see Figure 1.

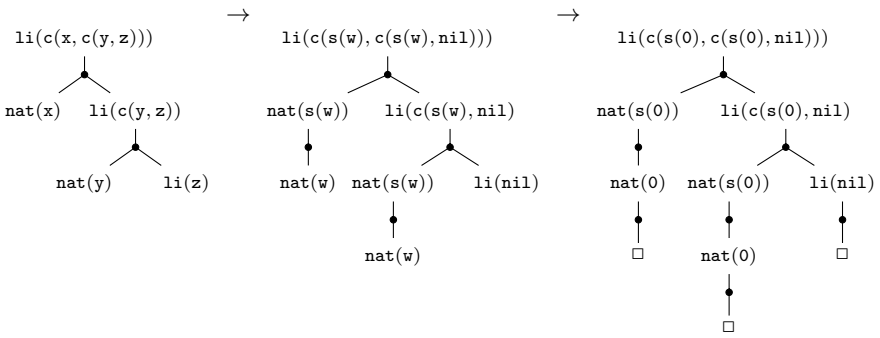

Fig. 1. Two derivation steps by coinductive derivation trees, for the program ListNat. We abbreviate `cons` by `c` and `list` by `li` in this figure. The symbol □ signifies "success". The last tree is a *success tree* which implies that the whole sequence of derivation steps above is successful.

3 Feature Selection and Vector representation

Pattern recognition methods [3] require feature selection and vector representation of data. One possible representation for sequential proofs is to use sequences of tactics similar to those considered in Example 2.

Example 3. In Example 2, we have considered a "positive" example of the tactic 4,1,4,1,5 for the goal G_0 = `list(cons(x,cons(y,x)))`. However, we cannot generalise this knowledge to future examples, as, the same tactic can produce an ill-formed derivation for the goal G_0 = `list(cons(x,cons(y,z)))`. G_1 = `nat(x), list(cons(y,x))`, G_2 = `list(cons(y,x))`, G_3 = `nat(y),list(z)`, G_4 = `list(z)`, G_5 = `fail`. The last goal `fail` makes the derivation ill-formed.

The example above shows that, having sufficiently big and representative set of training and testing examples, learning the tactics using this shallow method will not yield robust results. More generally, it appears that sequential algorithms lead to less natural machine learning realisations, due to the fact that they hide the structure of the proofs by allocating the main burden to sequential algorithms like unification and backtracking, [6, 5]. Compare the sequential derivations of Example 2 with coalgebraically-inspired coinductive trees. The latter does not employ either unification or backtracking, but have to handle derivations by exploiting structural properties of the trees - e.g., parallel branching; see [7] for a deeper analysis of this.

We will re-formulate the same task of proof-classification relative to coinductive trees. This time, we propose a more sophisticated feature selection method that captures the tree structure and patterns arising in these trees, e.g. dependencies between the structure of terms, predicates, and branching. For a given logic program P, a formula A, and the coinductive tree T built for A, we convert coinductive trees into vectors as follows.

1. Define a one-to-one function $|.|$ that assigns a numerical value to each function symbol in A, including nullary functions. Assign -1 to any variable occurring in T.

Example 4. For program ListNat, one encoding could be $|\texttt{0}| = 6$, $|\texttt{S}| = 5$, $|\texttt{cons}| = 2$, $|\texttt{nil}| = 3$, $|\texttt{x}| = |\texttt{y}| = |\texttt{z}| = -1$.

2. Complex terms are encoded by simple concatenation of the values of the function symbols. If the primitive value is negative, its positive value is concatenated, but the value of the whole term is negative.

Example 5. $|\texttt{cons(x,cons(y,x))}| = -21211$.

3. Build a matrix M for T as follows. The number of columns of M is equal to $n+2$, where n is the number of distinct predicates appearing in the program P. The number of rows is equal to the number m of distinct terms appearing in T. The entries of M are computed as follows. For the ith predicate R, and the jth term t, the ijth matrix entry is $|t|$ if $R(t)$ is a node of T, and 0 otherwise. For the $n+1$ column and the jth term t, if every node containing $Q(t)$ for some $Q \in P$ has exactly k children given by or-nodes, then the $(n+1)j$th entry in M is equal to k; and it is -1 otherwise. For the $n+2$ column and the jth term t, if all children of the node $Q(t)$, for some $Q \in P$ are given by or-nodes, such that all these or-nodes have children nodes $\Box$, then $(n+2)j$th entry is 1; if some but not all such nodes are $\Box$, then the then $(n+2)j$th entry is -1; and it is 0 otherwise.

Example 6. For the left-hand tree in Figure 1, the 4-by-5 matrix M is computed as shown inside the double lines:

	list	nat	•	□
cons(x, cons(y, z))	- 21211	0	2	0
cons(y, z))	- 211	0	2	0
x	0	-1	0	0
y	0	-1	0	0
z	-1	0	0	0

4. The matrix M is then flattened into a vector, so that, if the size of M is $(n+2) \times m$, then the vector V will have $(n+2) \times m$ elements.

Example 7. The matrix M above will be given by $V = [-21211, -211, -1, -1, 0, 0, 0, -1, 0, 0, 2, 2, 1, 1, 0, 0, 0, 0, 0, 0]$.

4 Machine-learning proof patterns

The obtained vectors are now suitable to be given as inputs to a neural network. All the experiments here were made in MATLAB Neural Network Toolbox (pattern-recognition package), with a standard three-layer feed-forward network, with sigmoid hidden and output neurons. Such networks can classify vectors arbitrarily well, given enough neurons in the hidden layer, we used 30 or 40 hidden layers for various experiments. The network was trained with scaled conjugate gradient back-propagation.

Problem 1. Classification of well-formed and ill-formed proofs. *Given a set of examples of well-formed and ill-formed coinductive trees, train the neural network so that, for any new example of a coinductive tree, it correctly classifies it in either of the two classes.*

Figure 1 shows three well-formed trees. Trees that do not conform to Definition 1 are ill-formed. A legitimate research question is: will this property yield pattern-recognition in neural networks? We used a set of 116 examples of well-formed and ill-formed trees, a sample of this set is given in the Appendix. The accuracy of classification reached as high as 73%.

Problem 2. Discovery of proof families. *Given a set of positive and negative examples of well-formed coinductive trees belonging to a proof family, train the neural network so that, for any new example of a coinductive tree, it correctly recognises whether it belongs to the given family.*

Definition 2. *Given a logic program P, and an atomic formula A, we say that a tree T belongs to the* family of coinductive trees determined by *A, if T is a coinductive tree with root A' and there is a substitution θ such that $A\theta = A'$.*

Example 8. The three trees in Figure 1 belong to the family of proofs determined by `list(cons(x,cons(y,z)))`.

Determining whether a given tree belongs to a certain coinductive family has practical applications. For Figure 1, knowing that the right-hand tree belongs to the same family as the left-hand-side tree would save the intermediate derivation step. Moreover, in [7], determining such intermediate tree required unification algorithm, which was the only part that did not yield parallelisation.

Machine learning offers an elegant solution to the problem. For the pattern recognition tool, we used 60 examples of trees classified as positive or negative examples for the family of trees for `list(cons(x,cons(y,z)))`, see the appendix for a sample. The accuracy of the neural-network based recognition reached 100% for some training setting and samples, and never fell lower than 98%.

Problem 3. Discovery of potentially successful proofs. Definition 2 conveys the idea of coinductive derivations, but it may not exactly capture the common intuition of what a proof is. For example, a coinductive tree for `list(cons(x,cons(y,x)))` will be in the same proof-family as trees of Figure 1, however, the formula will never be proven, and indeed it is false. We say that a proof-family F is a *success family* if, for all $T \in F$, T generates a proof-family that contains a success tree.

Proposition 1. *Given a coinductive tree T, there exists a success family F such that $T \in F$ if and only if T has a successful derivation.*

We used neural-network pattern-recognition tool to recognise trees from the success family; and, bearing in mind logical complexity of the notion, its accuracy was astonishing, reaching 95%.

5 Conclusions and Further Work

We have developed a method to analyse the structure of first-order proofs in logic programming. The method is based on coalgebraic approach to proofs; and also includes a feature-extraction algorithm that translates proof trees into vectors. As is usual for statistical pattern recognition, a suitable method for feature selection determines the success at the stage of neural network training. The success of learning the patterns show that the method has potential. It can be used for recognition of well-formed and ill-formed proofs, as in Problem 1; recognition of families of proofs, as in Problem 2; or even distinguishing potentially successful proofs, as in Problem 3. The latter two have significance for concurrent implementation of coinductive derivations of [7]. Ultimately, these three applications can be extended to proofs in higher-order theorem provers, and help to further advance their automatisation.

References

1. A. d'Avila Garcez, K. B. Broda, and D. M. Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications*. Springer-Verlag, 2002.
2. L. de Raedt. *Logical and Relational Learning*. 2008.
3. R. Duda, P. Hart, and D. Stork. *Pattern Classification*. John Wiley, 2001.
4. L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning*. MIT Press, 2007.
5. E. Komendantskaya. Parallel rewriting in neural networks. In *Proceedings of ICNC'09, Madeira, 3-7 October*. INSTICC, 2009.
6. E. Komendantskaya. Unification neural networks: Unification by error-correction learning. *Logic Journal of IGPL*, 2010.
7. E. Komendantskaya and J. Power. Coalgebraic derivations in logic programming. In *CSL'11*, 2011.
8. J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

A Examples of the trees from the training set for Problem 1.

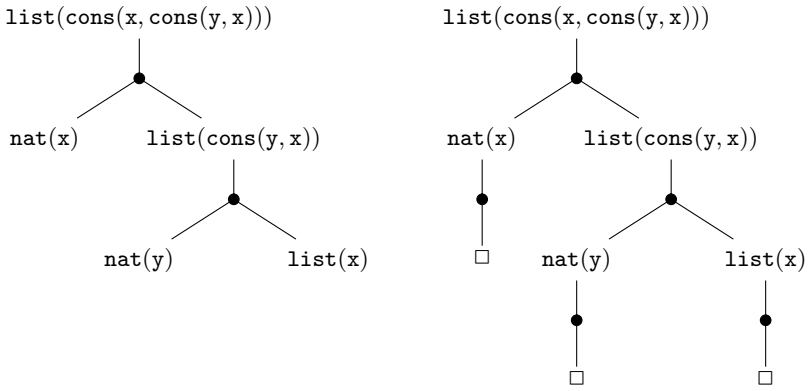

Fig. 2. An example of the training set for Problem 1. Left-hand-side tree is a positive example, and the right-hand-side — negative.

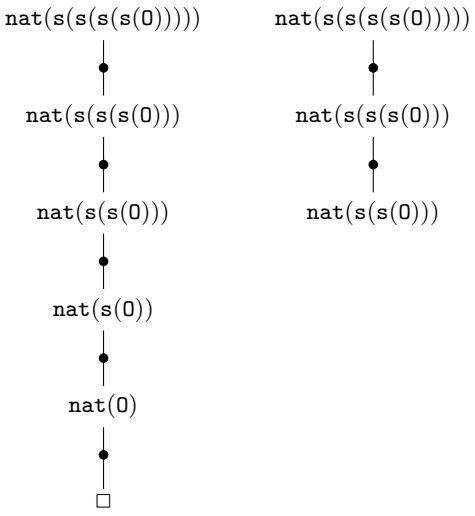

Fig. 3. An example of the training set for Problem 1. Left-hand-side tree is a positive example, and the right-hand-side — negative.

B An example of the training data set of proof trees for Problems 2 and 3.

Figure 1 shows three positive examples of the trees from the family determined by `list(cons(x,cons(y,z)))`. Here, we also show negative examples.

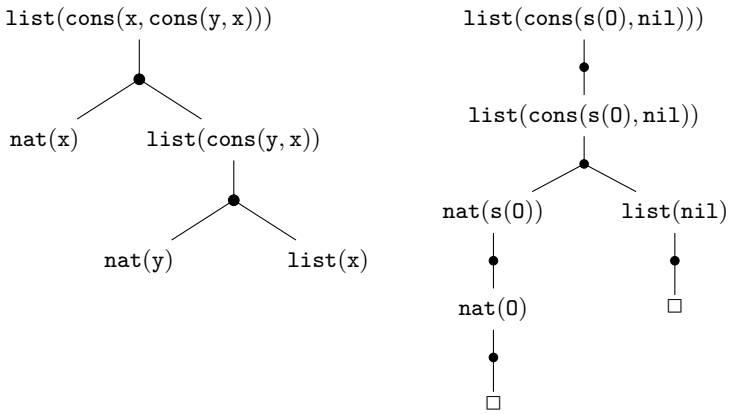

Fig. 4. The left-hand-side tree is a positive example for recognition of the trees belonging to the family of `list(cons(x,cons(y,z)))`, and the right-hand-side tree is not. However, both trees are negative examples of the trees from the success family determined by the formula. The positive examples of both family and success family of `list(cons(x,cons(y,z)))` are also given e.g. in Figure 1.