ReDuceDL: Inductive Programming by Greedy Folding of Certificates

Stephen Muggleton[1,2*]

[1*]School of Intelligence Science and Technology, Nanjing University, 163 Xianlin Avenue, Nanjing, 210023, Jiangsu Province, China.
[2]Department of Computing, Imperial College London, 180 Queens Gate, London, SW7 2AZ, United Kingdom.

Corresponding author(s). E-mail(s): s.muggleton@gmail.com;

Abstract

This paper describes a new learning system, ReDuceDL, which combines top-down and bottom-up Inductive Logic Programming (ILP) components from DeepLog and ReDuce respectively. The combination enables efficient learning of Logic Programs from one or more examples. In particular, ReDuceDL combines DeepLog's mechanisms for compilation of minimal certificates with ReDuce's published efficient greedy folding mechanism for identifying an underlying program from the certificates. This leads to linear-time learning and decomposition of multi-layered programs. Generalisation can be achieved from positive examples in the case that target programs are compact low generality solutions. The existing DeepLog paper demonstrated an error bound for a Bayesian framework in which low generality programs can be learned with low expected error from a single positive example. ReDuceDL employs DeepLog's compilation stage in which certificates associated with examples are generated, followed by ReDuce's interpretation stage which identifies minimal programs from the certificates. Consequently ReDuceDL achieves considerably higher efficiency than DeepLog when greedily folding certificates into logic programs, achieving speedups of more than 400 fold in some cases compared with DeepLog.

Keywords: Machine Learning, Mathematical Logic, Automated Programming, Greedy Folding

1 Introduction

This paper describes the integration of the author's top-down and bottom-up Inductive Logic Programming (ILP) systems DeepLog [18] and ReDuce [19] respectively. The resulting ReDuceDL system replaces DeepLog's non-linear time hypothesis space search by ReDuce's linear-time greedy folding of these certificates[1] into compact multi-layered logic programs.

We start by introducing some motivating examples. The relationship between Recursively Enumerable (RE) sets [3] and Universal Turing Machines (UTMs) [29] is foundational to Computer Science [25]. A set is recursively enumerable if it is recognized (or accepted) by a Turing machine, and a Universal Turing Machine is a specific machine that can recognize any RE set when provided with its description.

RE sets consist of of elements (typically integers or strings) which can be enumerated by a Turing machine, potentially running forever if the set is infinite. Also known as Turing-recognizable or semi-decidable, these sets can be verified if an element is present, but the Turing machine will fail to halt otherwise.

1.1 Inducing Recursively Enumerable sets

Question 1.1: Is it possible to learn a Recursively Enumerable set (RE set) efficiently from a single example?

The following simple example involves inducing a Recursively Enumerable (RE) set from a single element of the set.

Example 1 Assume $RE1(n) = \{2^n : n \in \mathbf{N}\}$. The unit set $RE1(11) = \{2048\}$. This can be demonstrated by using DeepLog to conduct a top-down derivation of a certificate, consisting of a sequence of functions as follows. $C(1, 2048) = \langle times2, times2, times2, times2, times2, times2, times2, times2, times2, times2, times2\rangle$. This certificate is then passed to ReDuce which conducts bottom-up greedy folding leading to the compact logic program below which can in turn enumerate $RE1(n)$ as re1(1,Y) where $Y \in \{1, 2, 4, ..2048, 4096, ..\}$.

```
re1(X,Y) :- star_times2(X,Z), eq(Z,Y).
star_times2(X,Y) :- eq(X,Z), eq(Z,Y).
star_times2(X,Y) :- times2(X,Z), star_times2(Z,Y).
```

Example 2 A more complex RE set is $RE2(n) = \{(2^{2^n} - 1) : n \in \mathbf{N}\}$ in which the unit set is $RE2(7) = \{r7\} = \{340282366920938463463374607431768211455\}$. DeepLog conducts a top-down derivation of the certificate $C(1, r7) = \langle plus1, square, square, square, square, square, square, square, minus1\rangle$. ReDuce then

[1] A Turing machine certificate (or "witness") is additional information provided to a polynomial-time verifier (a deterministic Turing machine) that proves a specific input belongs to a language in the complexity class NP. In this paper it takes the form of a sequence of primitive actions through which the input is transformed to the output.

uses bottom-up greedy folding to generalise the certificate resulting in the compact logic program below, which can then enumerate RE2(n) as re2(1,Y) where $Y \in \{1, 3, 15, 255, .., 340282366920938463463374607431768211455, ..\}$[2].

```
re2(X,Y) :- plus1(X,Z), re2_1(Z,Y).
re2_1(X,Y) :- star_square(X,Z), minus1(Z,Y).
star_square(X,Y) :- eq(X,Z), eq(Z,Y).
star_square(X,Y) :- square(X,Z), star_square(Z,Y).
```

1.2 Inducing Action models

Question 1.2: Is it possible, in time linear in the certificate sequence length, to identify a compact structured program?

The following example was previously introduced in [19].

Example 3 Imagine trying to devise a general fire escape program for evacuating a building. The program must ensure everyone in a bedroom exits the building via the shortest path. The DeepLog top-down derivation identifies a 191 length certificate $C(at(8, 8, 128), at(10, 10, 1)) = \langle fw, fw, .., fs, fs, .., fe, fe \rangle$. ReDuce then inductively generalises it to the compact logic program shown in Figure 1[3]. This shows the layout of the top and bottom floors of a building. An example certificate is provided which indicates a minimal escape sequence consisting of 191 individual actions {d,e,n,s,w} from position A on Floor 128 to the Exit position X on Floor 1. Given this sequence ReDuce[4] Figure 1 hypothesises a general hierarchical program which generates a minimal path sequence from any of the bedroom positions A,B,C on floors 2-128 to the exit position X. The timings (Figure 1d) indicate hypothesis generation for *ReDuce* strongly outperforms *DeepLog* and increases at most linearly with building height.

Note that ReDuceDL identifies a program, involving multiple recursions and introduction of new predicates (see Figure 1), from a single example However, since DeepLog search time increases non-linearly with sequence length, it only effectively identifies programs for buildings of at most 16 storeys high.

From example 2 and 3 above the author notes that Logic Programs, like those above, form a bridge between enumerative RE sets and Turing machines since the instantaneous state of a Turing machine, consisting of 0's and 1's, can be represented by a single natural number.

1.3 Organisation of paper

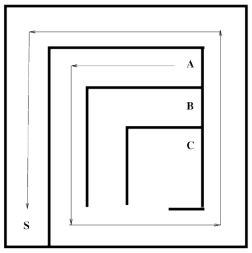
This paper is organised as follows. In Section 2 we review relevant work related to learning from one example, or *One-shot Learning*, in both Cognitive Science and Artificial

[2]Respective times taken for Example 1 and 2 are DeepLog compilation (0.04,0.06s) and ReDuce greedy folding (0.003s,0.003s). These were both based on SWI Prolog runnning on a single core 11th Gen Intel(R) i7-1165G7 @ 2.80GHz.

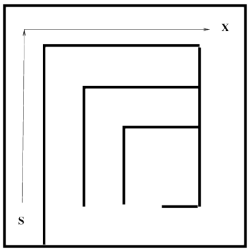
[3]Timings in Figure 1) for DeepLog and ReDuce were both based on SWI Prolog runnning on a single core 11th Gen Intel(R) i7-1165G7 @ 2.80GHz.

[4]*ReDuce* is named after the author's earlier *Duce* system [15] (see Section 2.1) which uses information reduction to guide a greedy search for hypotheses.

a) Floorplan



FLOOR 1



FLOOR 128

b) Hypothesised program (23 clauses):

```
f128(X,Y) :- defn_1(X,Z),f128_1(Z,Y).
f128_1(X,Y) :- star_e(X,Z),f128_2(Z,Y).
f128_2(X,Y) :- star_n(X,Z),f128_3(Z,Y).
f128_3(X,Y) :- defn_1(X,Z), f128_4(Z,Y).
f128_4(X,Y) :- star_d(X,Z), f128_5(Z,Y).
f128_5(X,Y) :- star_n(X,Z), star_e(Z,Y).

defn_1(X,Y) :- star_w(X,Z),star_s(Z,Y).

star_d(X,Y) :- eq(X,Z), eq(Z,Y).
star_d(X,Y) :- d(X,Z), star_d(Z,Y).
star_e(X,Y) :- eq(X,Z), eq(Z,Y).
star_e(X,Y) :- e(X,Z),star_e(Z,Y).
..
d([d|T],T).
e([e|T],T).
..
eq(X,X).
```

c) Certificate sequence (185 steps):

$\langle w, w, .., s, s, .., e, e, ..n, n, .., w, w, .., s, s, .., d, d, .., n, n, .., e, e, .. \rangle$

d) Timings

Floors	16	32	64	128
DeepLog	1.96s	-	-	-
ReDuceDL	0.05s	0.05s	0.05s	0.05s
Ratio	39.2	-	-	-

Fig. 1 a) Floorplan 2-128 have identical structure, "S" is Stair, "X" is Exit, A-C are starting bedroom positions. b) The hypothesised program is generated from c) the certificate sequence, provided by DeepLog, of 185 step actions from {d,e,n,s,w}. d) Hypothesis generation timings with 16, 32, 64 and 128 floors. DeepLog fails with more than 16 floors. By contrast, with repeated doubling in the number of floors ReDuceDL maintains a constant overall time up to 128 floors.

Intelligence. Following this, Section 3 introduces a mathematical framework consisting of a learning protocol, a Bayes' optimal solution and an associated Expected-Error bound for One-shot Learning. Section 4 describes the new ReDuceDL system, its relationship to DeepLog and ReDuce, and its supports for both one-shot and few-shot learning of efficient algorithms from relational examples. ReDuceDL achieves this by identifying algorithms which combine low description length with low generality. ReDuceDL is used in the experiments described in Section 5, which indicate that high accuracy algorithms can be efficiently learned from one example, not only for simple memory-free automata, but also for cases such as the textual analogy (Figure 2), in

which a push-down stack, of unbounded size, is required. In Section 6 we conclude and discuss possible areas for future research on this topic.

2 Related work

2.1 Duce - Propositional structure learning by compression

Duce [15] is a Machine Learning system which introduces propositional domain features on the basis of a set of examples. Duce uses six transformation operators to successively compress the examples by generalisation and feature introduction. Duce's main achievement was the restructuring of a substantial expert system for deciding whether positions within the chess endgame of King-and-Pawn-on-a7 versus King-and-Rook (KPa7KR) are won-for-white or not.

2.2 ReDuce - Bottom-up structure learning by compression

The ReDuce system [19] is similar to Duce in its use of successive compressive operations on sequences to introduce a structured hierarchy. By contrast, the ReDuce system constructs first-order recursive logic programs, allowing predicate arguments to carry and revise the state (eg the agent's position in the fire example in Section 1.2) during execution of the program.

2.3 DeepLog - Top-Down Meta-Interpretive Learning (MIL)

DeepLog[5],[6] [18] is an Inductive Logic Programming (ILP) [16, 22, 4] system, implemented in SWI-Prolog, based on MIL [20, 21, 24]. ILP systems hypothesise a relational logic program H from background knowledge B, primitive relations library $B_0 \subseteq B$, positive examples E^+ and negative examples E^-. MIL systems additionally require the user to provide a set of metarules M which indicate the structure of rules to be learned. By contrast, DeepLog minimises the requirements on the user by providing background knowledge from an existing library of definitions and uses a fixed metarule set M consisting of only one of the metarules from the Metagol system [20], namely *Chain* (Figure 5). The chain rule allows the introduction of a relation R based on relational composition of S and T. DeepLog [18] was the first paper to provide a theoretical bound on the generality of concepts which can be learned from a single positive example. This approach advanced the theoretical basis of one-shot learning beyond previous work in Cognitive Science and Linguistics and PAC learning models.

alice	ECILA
bert	**?**

Fig. 2 Analogy problem.

2.4 Cognitive Science

Consider a typical IQ-test question, based on textual analogy (see Figure 2), which says that "alice" is to "ECILA" as "bert" is to "?". In this case

[5] "Deep" refers to the initial deep dive to find a consistent certificate.
[6] DeepLog code and datasets available at https://github.com/StephenMuggleton/DeepLog/ .

assume the word indicated by "?" is "TREB" (Figure 3) since the relation R between "alice" and "ECILA" is "the reverse of the uppercase of the given letter sequence".

If there are no limits on the length of the letter sequences, an infinity of alternative responses might have been possible, including "ECILA". But for such an IQ-test question to be effective, the answer "TREB" must have high consensus among typical participants. In this paper it will be shown that standard Computational Learning Theory [7, 30] approaches do not account for humans' ability to hypothesise such a relation, R, from a single example.

alice	ECILA
bert	**TREB**

Fig. 3 Expected response.

Question 3: How can we model human abilities to hypothesise R from a single example?

An answer to this question has potential to provide insights into human perception and reasoning, and could help to identify new methods for efficient human-machine interaction. One relevant point worth noting is that human subjects presented with the problem are likely to already know the relevant commonly understood sub-concepts of *uppercase* and *reverse sequence*. However, it is unclear how a person could identify these two relations as the most relevant out of a large number of candidate relations which they already know.

To address the question above a modified version of an existing Computational Learning Theory approach [17] was developed in [18]. In the process, it was shown that for high Expected-Accuracy to be achieved on unseen cases, the Bayes' optimal hypothesis selected needs to provide a trade-off between the description length and generality of the algorithm representing R. Additionally, in the case of learning from a single example it is shown that the algorithm being learned needs to have low generality, meaning the chances are low for it correctly predicting the truth value of a randomly selected instance. This model of highly data efficient concept learning has ramifications concerning the lifetime learning of concepts, such as R, by humans having access to an accumulated mental library of previously learned low generality relations.

Over the last two decades there have been an increasing number of papers (e.g. [28, 13, 12, 31, 26]) demonstrating machine learning algorithms which learn concepts from a single example. This is referred to as "one-shot learning". Such algorithms are motivated by the observation that human learning often involves generalisations from a single example. However, such work has, to date, lacked mathematical analysis of the error associated with this form of learning. According to one of these papers published in the Cognitive Science literature [12] "People can learn .. concepts from just one example , but it remains a mystery how this is accomplished."

2.5 Linguistics

It is clear that some of the learning bias which enables one-shot learning comes from concepts already known to the human learner. However, it is not immediately obvious how many such background concepts humans have which might play a part in learning

a new concept. One bound on available concepts comes from studies in linguistics on the typical size of human vocabulary. According to one such study [8] the average adult knows somewhere in the range of 10,000 to 42,000 words, including, in our "alice" example case, "uppercase" and "reverse". However, any algorithm employing tens of thousands of background concepts would be not only overwhelmed by the available combinations of these terms, but also have to deal with the danger of over-fitting the example provided (e.g. always predicting "ECILA"). Despite this, humans are able to rapidly and reliably identify the relevance of "uppercase" and "reverse" and use these to hypothesise a new concept in the "alice/ECILA" example.

2.6 Identification in the limit

Computational Learning Theory (CoLT) [11, 1] is the study of algorithms which learn hypotheses from examples. The earliest such theoretical framework was introduced in 1967 by Mark Gold [6], and is known as *Identification in the Limit*, in which learning algorithms which are provided with a finite enumeration of both hypotheses and examples of a target language, for positing consistent hypotheses. Learning is effective in the case there exists a finite prefix of the example sequence after which the learning algorithm chooses the target language, and does not subsequently alter its hypothesis. Gold proved that, in general, for infinite formal languages (such as regular and context-free languages in the Chomsky hierarchy) such identification is not possible when learning from positive examples alone. The reason, as shown in Figure 4, is that for such language classes, given a finite number of examples, at no point can an algorithm discriminate between the most general language, consisting of all possible sequences from a given alphabet, or alternatively the most specific language, containing only the examples provided so far. Gold [6] pointed out the apparent disparity between this formal result and existing Psycholinguistic studies by McNeill [14], which had shown that children learn language largely from positive examples, and tend to ignore corrections to their use of grammar.

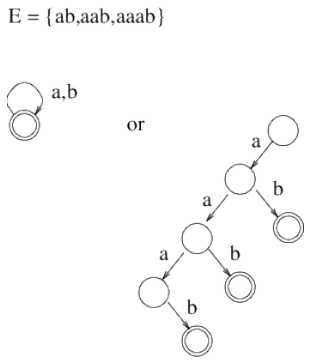


Fig. 4 Identification in the limit. Complexity versus generality.

2.7 Probably Approximately Correct (PAC) Learning

In 1984 a new framework for computational learning was introduced by Leslie Valiant [30]. By contrast to Gold's requirement of exact identification, PAC addresses learning algorithms which converge efficiently, with high probably, on an arbitrarily close approximation of the target theory, as increasing numbers of randomly selected examples are provided. Among those hypothesis classes considered by Valiant [30] only

k-CNF propositional logic formulae, has a PAC convergence proof based on positive examples alone. However, for a given propositional signature a k-CNF formula has a finite domain, as opposed to the grammar classes with infinite domains studied by Gold. In general, positive learnability results for the PAC framework tend to be restricted to highly constrained hypothesis classes, and the upperbounds on out-of-sample error tend to be weak when compared to the actual error in experimental trials.

2.8 Bayesian Positive-only Learning

In 1996 [17] the author introduced a framework to analyse the learning of logic programs from positive-only examples [7]. A Bayesian prior over the hypothesis space is assumed. The next section shows how a Bayesian approach allows the identification of Maximal Aposterior Probably (MAP) hypotheses which provide a tradeoff between the complexity of the hypothesis and its generality. It is shown that polynomial time logic programs can be learned with high accuracy from a randomly selected positive example sequence. This result goes beyond the positive-only results of Gold and Valiant, and supports efficient learning of infinite languages and programs. However, [17] falls short of providing effective error bounds for one-shot learning. We will address this issue in the next section.

3 ReDuceDL framework

3.1 Logical notation

A variable is represented by an upper case letter followed by a string of lower case letters and digits. A function symbol is a lower case letter followed by a string of lower case letters and digits. A predicate symbol is a lower case letter followed by a string of lower case letters and digits. The set of all predicate symbols is referred to as the predicate signature and denoted $\mathcal{P}$. An arbitrary reference total ordering over the predicate signature is denoted $\succeq_{\mathcal{P}}$. The arity of a function or predicate symbol is the number of arguments it takes. A constant is a function or predicate symbol with arity zero. The set of all constants is referred to as the constant signature and denoted $\mathcal{C}$. An arbitrary reference total ordering over the constant signature is denoted $\succeq_{\mathcal{C}}$. Functions and predicate symbols are said to be monadic when they have arity one and dyadic when they have arity two. Variables and constants are terms, and a function symbol immediately followed by a bracketed n-tuple of terms is a term. A variable is first-order if it can be substituted for by a term. A variable is higher-order if it can be substituted for by a predicate symbol. A predicate symbol or higher-order variable immediately followed by a bracketed n-tuple of terms is called an atomic formula or atom for short. The negation symbol is $\neg$. Both A and $\neg A$ are literals whenever A is an atom. In this case A is called a positive literal and $\neg A$ is called a negative literal. A finite set (possibly empty) of literals is called a clause. A clause represents the disjunction of its literals. Thus the clause $\{A_1, A_2, ..\neg A_i, \neg A_{i+1}, ...\}$ can be equivalently represented

[7] Later Tenenbaum [28] investigated positive-only learning in a Bayesian framework and referenced [17] but concentrated on an approach based on finite extensions. Note that infinite extensions, such as those considered in [17] and Section 3.5 of this paper, are required for learning algorithms such as *reverse*.

as $(A_1 \vee A_2 \vee ..\neg A_i \vee \neg A_{i+1} \vee ...)$ or $A_1, A_2, .. \leftarrow A_i, A_{i+1}, ...$ A Horn clause is a clause which contains at most one positive literal. A Horn clause is unit if and only if it contains exactly one literal. A denial or goal is a Horn clause which contains no positive literals. A definite clause is a Horn clause which contains exactly one positive literal. The positive literal in a definite clause is called the head of the clause while the negative literals are collectively called the body of the clause. A unit clause is positive if it contains a head and no body. A unit clause is negative if it contains one literal in the body. A set of clauses is called a clausal theory. A clausal theory represents the conjunction of its clauses. Thus the clausal theory $\{C_1, C_2, ...\}$ can be equivalently represented as $(C_1 \wedge C_2 \wedge ...)$. A clausal theory in which all predicates have arity at most one is called monadic. A clausal theory in which all predicates have arity at most two is called dyadic. A clausal theory in which each clause is Horn is called a logic program. A logic program in which each clause is definite is called a definite program. Literals, clauses and clausal theories are all well-formed-formulae (wffs) in which the variables are assumed to be universally quantified. Let E be a wff or term and σ, τ be sets of variables. $\exists\sigma.E$ and $\forall\tau.E$ are wffs. E is said to be ground whenever it contains no variables. E is said to be higher-order whenever it contains at least one higher-order variable or a predicate symbol as an argument of a term. E is said to be datalog if it contains no function symbols other than constants. A logic program which contains only datalog Horn clauses is called a datalog program. The set of all ground atoms constructed from $\mathcal{P}, \mathcal{C}$ is called the datalog Herbrand Base. $\theta = \{v_1/t_1, .., v_v/t_n\}$ is a substitution in the case that each v_i is a variable and each t_i is a term. $E\theta$ is formed by replacing each variable v_i from θ found in E by t_i. μ is called a unifying substitution for atoms A, B in the case $A\mu = B\mu$. We say clause C θ-subsumes clause D or $C \succeq_\theta D$ whenever there exists a substitution θ such that $C\theta \subseteq D$.

3.2 Framework

We first define the higher-order *meta-rules* used by the Prolog meta-interpreter.

Definition 1 (Meta-rules) A meta-rule is a higher-order wff

$$\exists\sigma\forall\tau P(s_1, .., s_m) \leftarrow .., Q_i(t_1, .., t_n), ..$$

where σ, τ are disjoint sets of variables, $P, Q_i \in \sigma \cup \tau \cup \mathcal{P}$ and $s_1, .., s_m, t_1, .., t_n \in \sigma \cup \tau \cup \mathcal{C}$. Meta-rules are denoted concisely without quantifiers as

$$P(s_1, .., s_m) \leftarrow .., Q_i(t_1, .., t_n), ..$$

The quantified version of the meta-rules in Figure 5 is shown in Figure 5. In general, unification is known to be semi-decidable for higher-order logic [10]. We now contrast the case for higher-order datalog programs.

Proposition 1 (Decidable unification) *Given higher-order datalog atoms $A = P(s_1, .., s_m)$, $B = Q(t_1, .., t_n)$ the existence of a unifying substitution μ is decidable.*
Proof. A, B has unifying substitution μ iff $p(P, s_1, .., s_m)\mu = p(Q, t_1, .., t_n)\mu$.

Name	Meta-Rule	Quantified version
Instance	$P(X,Y) \leftarrow$	$\exists PXY\ P(X,Y)$
Base	$P(x,y) \leftarrow Q(x,y)$	$\exists PQ \forall xy\ P(x,y) \leftarrow Q(x,y)$
Chain	$P(x,y) \leftarrow Q(x,z), R(z,y)$	$\exists PQR \forall xyz\ P(x,y) \leftarrow Q(x,z), R(z,y)$
TailRec	Same as Chain	

Fig. 5 Quantification of meta-rules in Figure .

Meta-Substitution	Higher-order substitution
$metasub(instance, [father, ted, bob])$	$\{P/father, X/ted, Y/bob\}$
$metasub(base, [ancestor, p1])$	$\{P/ancestor, Q/p1\}$
$metasub(tailrec, [ancestor, p1, ancestor])$	$\{P/ancestor, Q/p1, R/ancestor\}$

Fig. 6 Relationship between meta-substitutions used by the meta-interpreter and higher-order substitutions.

Generalised meta-interpreter

```
prove([], Prog, Prog).
prove([Atom|As], Prog1, Prog2) :-
    metarule(Name, MetaSub, (Atom :- Body), Order),
    Order,
    save_subst(metasub(Name, MetaSub), Prog1, Prog3),
    prove(Body, Prog3, Prog4),
    prove(As, Prog4, Prog2).
```

Fig. 7 Prolog code for the generalised meta-interpreter. The interpreter recursively proves a series of atomic goals by matching them against the heads of meta-rules. After testing the Order constraint *save_subst* checks whether the meta-substitution is already in the program and otherwise adds it to form an augmented program. On completion the returned program, by construction, derives all the examples.

Figure 6 shows for the meta-rules from Figure 5 of the relationship between the meta-substitutions constructed by the meta-interpreter (Figure 7) and higher-order substitutions of existential variables.

Definition 2 (MIL setting) Given meta-rules M, definite program background knowledge B and ground positive and negative unit examples E^+, E^-, MIL returns a higher-order datalog program hypothesis H if one exists such that $M, B, H \models E^+$ and M, B, H, E^- is consistent.

The following describes decidable conditions for MIL.

Theorem 2 (MIL decidable) *The MIL setting is decidable in the case M, B, E^+, E^- are Datalog and $\mathcal{P}, \mathcal{C}$ are finite.*
Proof. Follows from the fact that the set of Herbrand interpretations is finite.

3.3 Language classes and expressivity

We now consider whether a variant of MIL could be formulated within the Learning from Interpretations setting [5] in which examples are pairs $\langle I, T \rangle$ where I is a set of ground facts (an interpretation) and T is a truth value. In this case each interpretation I will be a subset of the datalog Herbrand Base, which is constructed from $\mathcal{P}$ and $\mathcal{C}$. To account for predicate invention we assume that $\mathcal{P}' \subseteq \mathcal{P}$ and $\mathcal{C}' \subseteq \mathcal{C}$ represent uninterpreted predicates and constants respectively, whose interpretation is assigned by the Meta-interpreter. Since the user has no ascribed meaning for $\mathcal{P}'$ and $\mathcal{C}'$, it would not be possible to provide examples containing full interpretations, and therefore the Learning from Interpretations setting is inappropriate. Clearly, a variant of the Learning from Interpretations setting could be introduced in which examples are represented as incomplete interpretations.

3.4 Language classes and expressivity

We now define language classes for instantiated hypotheses.

Definition 3 (H^i_j program class) Assuming i, j are natural, the class H^i_j contains all higher-order definite datalog programs constructed from signatures $\mathcal{P}, \mathcal{C}$ with predicates of arity at most i and at most j atoms in the body of each clause.

The class of dyadic logic programs with one function symbol has Universal Turing Machine (UTM) expressivity [27]. Note that H^2_2 that the fragment also has UTM expressivity, as demonstrated by the following H^2_2 encoding of a UTM in which $S, S1, T$ represent Turing machine tapes.

```
utm(S,S) ← halt(S).
utm(S,T) ← execute(S,S1), utm(S1,T).
execute(S,T) ← instruction(S,F), F(S,T).
```

We assume the UTM has a suitably designed set of machine instructions representing functions of the form

$$f : \mathcal{T} \to \mathcal{T}$$

where $\mathcal{T}$ is the set of all Turing machine tapes. Below assume G is a datalog goal and program $P \in H^2_2$.

Proposition 3 (Undecidable fragment of H^2_2) *The satisfiability of G, P is* undecidable *when $\mathcal{C}$ is infinite.*
Proof. Follows from undecidability of halting of UTM above.

The situation differs in the case $\mathcal{C}$ is finite.

Theorem 4 (Decidable fragment of H_2^2) *The satisfiability of* G, P *is* decidable *when* $\mathcal{P}, \mathcal{C}$ *is finite.*
Proof. The set of Herbrand interpretations is finite.

The universality of the H_2^2 fragment is established by the existence of the UTM above. However, there are many concepts for which this approach would lead to a cumbersome and inefficient approach to learning. For instance, consider the following definition of an undirected edge.

$$undirected(A, B) : -edge(A, B), edge(B, A).$$

A program for the UTM above would be required to search through each pair of edges to find a pair of nodes A, B with associated edges in each direction. As a deterministic program this is non-trivial and is likely to involve two iterative loops. However, the clause above is directly and compactly representable within H_2^2 given the following meta-rule.

$$P(x, y) \leftarrow Q(x, y), R(y, x)$$

Thus effective use of meta-rules can lead to a more constrained and effective search. Although the meta-interpreter can be applied to meta-rules containing more than two atoms in the body, we limit ourselves to the H_2^2 fragment throughout this paper. This restriction limits the number of possible meta-rules and forces more prolific predicate invention, leading to more opportunities for internal re-use of part of the hypothesised program.

3.5 DeepLog framework

3.5.1 Bayesian Learning Protocol

For the purposes of this paper we introduce a specialisation of the teacher-learner positive-only protocol introduced in [17]. In this variant, relational instances are atoms $r(a, b)$, where r is a relation (i.e. predicate of arity 2) and a and b are ground terms[8]. A relational program is a logic program in which all predicates have arity 2.

- X is a countable[9] set of relational instances.
- D_X is the teacher's probability distribution over X.
- $\mathcal{H} \subset 2^X$ is a countable hypothesis set for which each $H \in \mathcal{H}$ represents the least Herbrand model of a relational program.
- $D_{\mathcal{H}}$ is the teacher's probability distribution over $\mathcal{H}$.
- The teacher randomly chooses target theory $T \in \mathcal{H}$ from $D_{\mathcal{H}}$ and then chooses $E = x_1 \dots x_m$ randomly and independently from $D_{X|T}$,with probability $x \in E$ such that

$$D_{X|T}(x) = D_X(x|T) = \frac{D_X(x \cap T)}{D_X(T)} = \begin{cases} 0 & \text{if } x \notin T \\ \frac{D_X(x)}{D_X(T)} & \text{otherwise} \end{cases}$$

[8] For instance, X = {rvu(⟨⟨A,l,i,c,e⟩, ⟨E,C,I,L,A⟩⟩), rvu(⟨⟨B,e,r,t⟩, ⟨T,R,E,B⟩⟩),...} are relational instances of the relation *rvu*. Note that an algorithm like reverse-uppercase is usually defined over a countably infinite set, and is both a relation and a one-to-one function.
[9] Either finite or countably infinite.

$$
\begin{aligned}
p(H|E) &= \tfrac{p(H)p(E|H)}{p(E)} \ \textbf{[Bayes theorem]} \\
&= \frac{p(H)\prod_{i=1}^{m}\frac{D_X(x_i)}{g(H)}}{p(E)} \\
&= p(H)\left(\tfrac{1}{g(H)}\right)^m c_m \\
\\
-ln\ p(H|E) &= sz(H) + m\,(ln\ g(H)) + d_m \\
\\
\Longrightarrow\ &\textbf{Minimise}\ -ln\ p(H|E)\ \text{over}\ H \in \mathcal{H}
\end{aligned}
$$

Fig. 8 Positive-only MAP selection. $m = |E|$, $x_i \in E$, $c_m = \prod_{i=1}^{m} D_X(x_i)$ and $d_m = ln\ c_m$.

- The learner now selects $H \in \mathcal{H}$ for which $E \subseteq H$.
- For $H \in \mathcal{H}$, $D_X(H) = \sum_{x \in H} D_X(x)$.
- The teacher then assesses $Error(H, T)$ as $D_X(H \setminus T) + D_X(T \setminus H)$.
- The Expected-Error (EE) of hypothesis H is then

$$EE(m) = \sum_{T \in \mathcal{H}} D_{\mathcal{H}}(T) \sum_{x \in E \in T^m} D_X(x|T) Error(H, T).$$

  where the set of all cardinality m training sets T^m is defined recursively as follows. $T^1 = T$ and $T^m = T \times T^{m-1}$.
- $sz(H) = -lnD_{\mathcal{H}}(H)$ is referred to as the size of H.
- $g(H) = \sum_{x \in H} D_X(x)$ is referred to as the generality of H.

3.5.2 Positive-only MAP selection

Using the Bayesian Learning Protocol the learner's positive-only MAP selection method introduced in[17] is shown in Figure 8. The complexity versus generality problem highlighted by Gold's result (Figure 4) is resolved by the learner selecting an hypothesis H which trades-off $sz(H)$ and $g(H)$[10]. As the number of training examples m increases, the need to minimise $g(H)$ progressively dominates minimising $sz(H)$, which contrasts with the fixed hypothesis ordering assumption of Gold [6]. Let us now consider Bayesian one-shot learning.

$$-ln\ p(H|E) = sz(H) + \ ln\ g(H) + d_1$$

Fig. 9 Bayesian one-shot learning, $m = 1$ case.

[10] While [17] assumes $p(E|H) = \left(\frac{1}{g(H)}\right)^m$, Tenenbaum's *size principle* in [28] is that $p(E|H) = \left(\frac{1}{size(H)}\right)^m$. The former corresponds to the probability of H, with potentially *infinite extension*, being true of m instances randomly selected from D_X, while the latter is the non-normalised probability of concept H, with *finite extension* size(H), being true of m instances randomly selected from a uniform distribution over X.

3.5.3 Bayesian one-shot learning

One-shot learning is simply the special case of Bayesian positive-only learning for which $m = 1$ (see Figure 9). However, this case was not considered in [17], since the Expected-Error bound derived in that paper gives $EE(1) \leq 2.33$, which is trivially true since error is in the $[0, 1]$ interval.

Type	**Expected-Error**
Positive-only[17]	$EE(m) \leq \frac{2.33+2ln\ m}{m}$
Positive+Negative[17]	$EE(m) \leq \frac{1.51+2ln\ m}{m}$
One-shot	$EE(m = 1\|g(T)) \leq 4.66g(T)$

Fig. 10 Previous and new Expected-Error bounds.

3.5.4 Expected-Error bounds

Expected-Error results are shown in Figure 10. The positive-only result was based on the assumption that $g(T) \leq \frac{1}{2}$. Under this assumption, it was noted [17] that the bounds indicate (see Figure 10) that it does not take many more examples to learn from positive-only than from a mixture of positive and negative examples. This was found to be consistent with out-of-sample results for an implementation of these two approaches [17]. The third is a new result of the present paper. This can be derived by generalising the $g(T) \leq \frac{1}{2}$ assumption to $g(T) \leq \alpha$ where $0 \leq \alpha \leq 1$, in which case $EE(m) \leq \frac{\alpha(4.66+4ln\ m)}{m}$. In the one-shot learning case, $m = 1$, we get $EE(m = 1) \leq \frac{\alpha(4.66+4ln\ 1)}{1}$ and therefore $EE(m = 1|g(T)) \leq 4.66g(T)$.

$EE(m = 1|g(T)) \leq 0.05$ when $g(T) \leq 0.01$

Fig. 11 Low-generality/high-accuracy region.

Monkey	Banana
Sparrow	**Worm**

Fig. 12 Analogy involving relation *eats*.

3.5.5 Low-generality targets

The One-shot Expected-Error bound in Figure 10 allows us to predict high accuracy hypotheses are possible from one example if and only if the generality of the target hypothesis is low (see Figure 11).

Chain: $R(x,y) \leftarrow S(x,z), T(z,y)$

Fig. 13 Chain metarule defines relation R as composition of S and T. Only clauses of this form are included in programs hypothesised by DeepLog.

3.5.6 Choice of representation

The motivation for DeepLog [18] was to analyse and demonstrate a general approach to learning from a single example in a way that models human abilities. One-shot learning of an algorithm, is particularly challenging, since it involves a countably infinite domain (eg reversing and uppercasing a sequence such as $\langle a, l, i, c, e \rangle$). However, humans also have a capacity to identify relations from a single example, as part of a broad range of analogy problems. For instance, the answer "Insect" to the analogy problem in Figure 12 would be equally acceptable for a human to that given. For this reason, rather than select a functional programming language, the base representation is that of relational logic programs rather than functional programs, though the results in Figure 10 would apply in either case since functions can be considered as many-to-one relations.

3.6 ReDuce framework

3.6.1 Lossless compression

For the purposes of this paper we define *lossless compression* as follows.

Definition 4 Lossless compression Let σ represent a finite set of symbols, σ^* be the countable set of sequences of symbols from σ, $|s|$ be the length of a sequence s and κ_1^{-1} represent a function and its inverse over σ^*. We say that κ_1 is a *lossless compression* function iff $\forall s \in \sigma^*, t \in \sigma^*$ it is the case that $t = \kappa_1(s)$ iff both $s = \kappa_1^{-1}(t)$ and $|t| < |s|$.

Note that *lossless compression* can be used to reduce the length of a sequence (or text), and that the original sequence can be retrieved without loss of information. Sequitur [23] is a linear-time Lossless compression algorithm.

Lossless compression is usually contrasted with *lossy compression*, in which an approximation of the original text is retrieved.

3.6.2 Inductive compression

In this paper we consider a variant of lossy compression which is referred to as *inductive compression* since it can be viewed as a form of inductive inference.

Definition 5 Inductive compression Let σ, σ^* and $|s|$ be as in Definition 4. Additionally let set s be such that $s \in \sigma^*$ and $\kappa_2(s)$ is an encoding of s. We say that $\kappa_2(s)$ is an *inductive compression* of s iff $|\kappa_2(s)| < |s|$.

Example 4 Both Duce [15] and the ReDuce approach described in this paper are Inductive compression algorithms.

We now show that both Lossless compression and Inductive compression algorithms are finitely terminating.

Theorem 5 Finite termination *Assume A is either a Lossless or Inductive Compression algorithm, $s_0 \in \sigma^*$ is a finite sequence of length $|s_0|$. It follows that $s = A(s_0)$ finitely terminates after n applications of κ where $0 \leq n < |s_0|$.*

Proof Assume Theorem 1 is false. In the case $\kappa(s_0)$ fails, then $s_0 = A(s_0)$ finitely terminates with $n = 0$. So assume $\kappa(s_0)$ succeeds. In this case $s = A(s_0)$ finitely terminates after n applications of κ where $n \geq |s_0|$. However, each application of κ reduces the sequence by at least one, so it follows that $n < |s_0|$, which contradicts the assumption and completes the proof. □

3.6.3 Greedy inductive compression

An algorithm is referred to as *greedy* when locally optimal choices are selected at each stage. In the case of iterative sequence compression this does not guarantee a globally smallest solution, but efficiently produces a near-smallest solution. The ReDuce system described in the next section uses a greedy strategy which iteratively computes the frequency of neighbouring pairs of symbols in a given sequence S. In the case that a subsequence xy has maximum frequency in S of at least 2, then this pair is replaced throughout the sequence by a unique symbol, such as T, which is added as a definition $T \rightarrow xy$. In this paper we refer to this process as sequence folding. Example 5 shows how iterative greedy sequence folding works in lossless compression.

Example 5 Iterative greedy sequence folding. Consider the sequence $S \rightarrow aaabbbbbcccaabbcc$. The frequencies f of pairs of symbols in S are $f(aa) = 3, f(ab) = 2, f(bb) = 4, f(bc) = 1, f(cc) = 3$. A greedy choice for folding is $S \rightarrow aaaTTcccaaTcc, T \rightarrow bb$. When this folding process is iterated it terminates with $S \rightarrow UaTWcUW, T \rightarrow bb, U \rightarrow aa, V \rightarrow cc, W \rightarrow TV$.

3.6.4 Greedy inductive compression with star-repeat

Example 5 shows how folding supports introduction of nested structure, in the form of sub-expressions. Example 6 below shows how star-repeat expressions, used in Regular Expressions, can generalise sequences by replacing repeated symbols, such as aaa, by star-repeat expressions such as a^*.

Example 6 Inductive compression. Once more consider the sequence $S \rightarrow aaabbbbbcccaabbcc$. Using star-repeat expressions gives $S \rightarrow UVWUVW, U \rightarrow a^*, V \rightarrow b^*, W \rightarrow c^*$, which is an inductive generalisation of the sequence. Following this a greedy lossless compression step introduces further structure, giving $S \rightarrow XWXW, U \rightarrow a^*, V \rightarrow b^*, W \rightarrow c^*, X \rightarrow UV$. In a further lossless iteration this becomes $S \rightarrow YY, U \rightarrow a^*, V \rightarrow b^*, W \rightarrow c^*, X \rightarrow UV, Y \rightarrow XW$. Finally, a star-repeat generalisation produces $S \rightarrow Z, U \rightarrow a^*, V \rightarrow b^*, W \rightarrow c^*, X \rightarrow UV, Y \rightarrow XW, Z \rightarrow Y^*$. Note this grammar has multiple levels of both structure and iteration and cannot be compacted further.

4 ReDuceDL implementation

As already mentioned, ReDuceDL combines top-down and bottom-up Inductive Logic Programming (ILP) components from DeepLog and ReDuce respectively, to enable efficient learning of Logic Programs from one or more examples. In particular, ReDuceDL employs DeepLog to generate minimal certificates which are then passed to ReDuce which in turn conducts greedy folding to identify an underlying program from the certificates. This leads to linear-time learning and decomposition of multi-layered programs. Generalisation can be achieved from positive examples in the case that target programs are compact low generality solutions. The existing DeepLog paper demonstrated an error bound for a Bayesian framework in which low generality programs can be learned with low expected error from a single positive example.

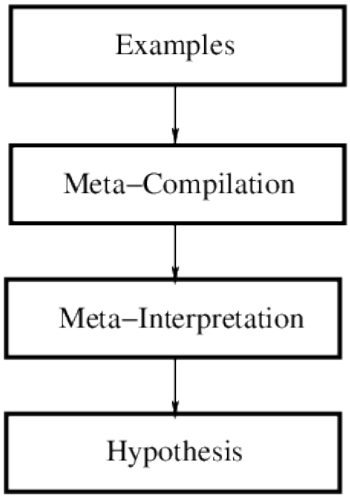


Fig. 14 DeepLog stages.

4.1 DeepLog

DeepLog[11][12] is an Inductive Logic Programming (ILP) [16, 22, 4] system, implemented in SWI-Prolog, based on Meta-Interpretive Learning (MIL) [20, 21, 32, 24]. ILP systems hypothesise a relational logic program H from background knowledge B, primitive relations library $B_0 \subseteq B$, positive examples E^+ and negative examples E^-. MIL systems additionally require the user to provide a set of metarules M which indicate the structure of rules to be learned. By contrast, DeepLog minimises the requirements on the user by providing background knowledge from an existing library of definitions and uses a fixed metarule set M consisting of only one of the metarules from the Metagol system [20], namely *Chain*, shown in Figure 13. The chain rule allows the introduction of a relation R based on relational composition of S and T.

4.1.1 DeepLog architecture

The overall structure of DeepLog is shown in Figure 14.

[11] "Deep" refers to the initial deep dive to find a consistent certificate.
[12] DeepLog code and datasets available at https://github.com/StephenMuggleton/DeepLog/ .

4.1.2 Meta-Compilation

The positive examples E^+ are used to find a set of minimal length input-output certificates[13], where C is a length n certificate of an example $r(a,b) \in E+$ if $r(a,b)$ is a relational instance, a and b are logical terms and there exists a sequence of ground instances $C = \langle p_0(a,x_0),..,\ p_n(x_n,b)\rangle$ (simplified below to $C = \langle p_0,..,\ p_n\rangle$). $B_0 \models \alpha$ for each ground atom α in C. C has minimal length n in case there is no certificate C' for $r(a,b)$ of length n' where $n' < n$. This step is used to identify both a minimal signature of primitive relations from the library B_0 and a minimal universe of logical terms x_i to be considered as intermediate states in the certificates considered by the Meta-Interpretation stage. Minimal (Min[14])[17] and maximal (Max[15]) bounds on the number of clauses in any hypothesised program are identified. Additionally, repeated subsequences of the minimal certificates are identified as potentially useful auxiliary primitives. Finally, a set of binary square matrices are computed to provide a look-up oracle for the Meta-interpreter to rapidly identify minimal certificates and optimal choice points in the meta-interpreter's derivation of hypotheses.

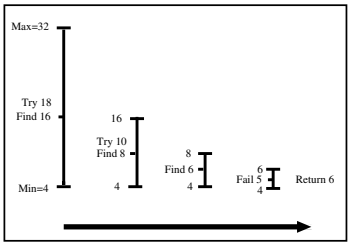


Fig. 15 Binary search exponentially reduces hypothesis space to return consistent program with minimal number of clauses.

4.1.3 Meta-Interpretation

A binary search procedure (Figure 15) is used to progressively reduce the $\langle Min, Max\rangle$ boundaries in order to find the minimum number of clauses in any consistent hypothesis returned by the meta-interpreter. For any specific $\langle Min, Max\rangle$ pair the meta-interpreter's search order considers more specific hypotheses before more general ones, and returns the first consistent hypothesis. This strategy is aimed at finding low complexity hypotheses which have low generality, in accordance with the Positive-only MAP selection strategy described in Section 3.5.

4.2 ReDuce implementation

Below is a top-level pseudo-code description of the ReDuce algorithm.

Algorithm 1 can be illustrated by considering Example 6 (Section 3.6.4). The Initial Grammar (step 1) is $S \rightarrow aaabbbbbcccaabbcc$. This is reduced by Star-Replacement

[13] In Computability theory these are called *certificates*, in Graph theory they are called *edge-labelled paths* and in the AI planning literature they are *sequential plans*.
[14] *Min* is the length of the minimal number of relations in any minimal certificate identified.
[15] *Max is one less than the length of a minimal certificate*

Algorithm 1 ReDuce(InitialGrammar)

1: $G \leftarrow$ StarReplace(InitialGrammar)
2: **while** Reducible(G) **do**
3: Pair in $G \leftarrow$ MaxFrequencyPairInBody(G)
4: $G \leftarrow$ DefineAndSubstituteAll(Pair,G)
5: G $\leftarrow$ StarReplace(G)
6: **end while**
7: **return**(G)

to $S \rightarrow UVWUVW$. This is Reducible (step 2) since both UV and VW have a frequency of at least 2. UV is then selected (step 3) and substituted (step 4) to give $S \rightarrow XWXW$. Star-Replacement has no effect (step 5) since no adjacent symbols are identical. After iterating (step 2) $XWXW$ is reducible since XW has frequency 2, and so (step 4) these are replaced to give $S \rightarrow YY$. On the following iteration this is reduced to $S \rightarrow Z$. This is no longer reducible and so (step 7) the revised grammar is returned together with all introduced definitions[16].

Theorem 6 Correctness of Algorithm 1 *By construction the initial sequence can be generated from the S rule in the grammar G by successive unfolding of symbols in the right-hand side.*

Proof Consider Algorithm 1. Let S_0 represent the initial grammar, $S_n = ReDuce(S_0)$ and $S_0, S_1, .., S_n$ represent the sequence of folds leading from S_0 to S_n. By construction there exists a sequence of unfold operations $S_n, S_n - 1, .., S_0$ which generate S_0 from S_n. This completes the proof. □

A critical motivation for this paper is the potential reduction in time-complexity required for hypothesis formation within ILP. We now address the time complexity of Algorithm 1.

Theorem 7 Time complexity *All pairs of symbols and their frequencies can be identified in time linear in the initial sequence.*

Proof Each cycle of Algorithm 1 reduces the right-hand-side of S by at least one symbol. Each such reduction requires constant time when conducted using hash-indexed substitutions. Thus the number of cycles is bounded by the length n of the right-hand-side of S, and there are at most n indexed foldings. This implies that Algorithm 1 terminates in $O(n)$ time and space. This completes the proof. □

Note this is the same order of complexity as Neville-Manning and Witten [23] result for the Sequitur algorithm, which is widely used for large-scale text compression.

[16] Note that Prolog definitions such as those in Figure 1 are generated from grammar rewrite rules such as $P \rightarrow QR$ using meta-rule templates such as $P(x, y) \leftarrow Q(x, z), R(z, y)$. The resulting Prolog rules contain variables, such as x, y, z, which allow State Transformations in the resulting program, enabling the learning of recursively enumerable functions such as *Reverse Uppercase* (see Problem 3 in Figure **??** below.

5 Experiments

5.1 DeepLog Experiments

In this subsection we review various results presented in the DeepLog paper [18] in the learning of three specific target programs. For each target program the same background library of 62 primitive relations was used. The library was developed progressively from investigating 57 different problems, and contains a set of type identifiers and basic relations over various types of entities, including characters, lists, stacks, numbers, 3D-positions, chess and family relations. In the problem descriptions below one-shot percentage Expected-Accuracy is defined as $EA(1,T) = 100(1 - EE(m = 1|g(T)))$.

Target	**Example** e^+	**Primitives** P
abc4	⟨a,b,c,d,e,f,c,d,e,f,g,h⟩ → ⟨⟩	Library 62 primitives

Minimal certificates	$\{\langle a,b,c,d,e,f,c,d,e,f,g,h\rangle\}$

Hypothesised Program H (7 clauses)	**Training**
abc4(X,Y) :- a(X,Z), abc4_1(Z,Y). abc4_1(X,Y) :- b(X,Z), abc4_1_1(Z,Y). abc4_1_1(X,Y) :- g(X,Z), h(Z,Y). abc4_1_1(X,Y) :- cdef(X,Z), abc4_1_1(Z,Y). **Auxiliary primitives** cdef(X,Y) :- cd(X,Z), ef(Z,Y). cd(X,Y) :- c(X,Z), d(Z,Y). ef(X,Y) :- e(X,Z), f(Z,Y).	$Time = 0.15s/0.56s$ $g(H) = \frac{1}{1080} \approx 0.0009$ $-ln\ p(H\|E) = 17.97$ $EA(1,T) > 99.57\%$

Fig. 16 Problem1: Example, minimal certificate, hypothesis, primitives used and training. Expected-Accuracy (EA) is defined as EA(1,T) = 100(1-EE(H)).Primitives *a,b,..,h* remove a single character from the input to produce the output. The input and output type *cl* is a *character list*. The hypothesised program represents the regular grammar $ab(cdef)^*gh$

.

Problem1: Regular Grammar

Consider a formal language which involves repeated letter sequences. For instance, the positive example sequence e^+ = "abcdefcdefgh" might be used to exemplify the target language G = ab(cdef)*gh. G corresponds to letter sequences, such as e^+, which have a prefix $\langle a,b\rangle$, a suffix $\langle g,h\rangle$ and zero or more repetitions of $\langle c,d,e,f\rangle$ inbetween. Figure 16 shows e^+ as an input/output pair and DeepLog's hypothesised Logic Program, corresponding to G. This program is constructed from primitives which reduce the sequence by one letter (such as a, b, ..), auxiliary relations (such as cdef, cd, ..) composed from the primitives introduced during Meta-Compilation (Figure 14), and

abc4(X,Y) :- a(X,Z), abc4_1(Z,Y). abc4_1(X,Y) :- b(X,Z), abc4_1_1(Z,Y). abc4_1_1(X,Y) :- g(X,Z), h(Z,Y). abc4_1_1(X,Y) :- cdef(X,Z), abc4_1_1(Z,Y).	u=g(abc4) v=g(abc4_1) w=g(abc4_1_1)

$u = \frac{v}{6},\ v = \frac{w}{6},\ w = \frac{1}{6}^2 + \frac{w}{6}$ $\implies u = \frac{w}{36},\ w = \frac{1}{36} + \frac{w}{6}$ $\implies \frac{5w}{6} = \frac{1}{36}$ $\implies w = \frac{6}{180} = \frac{1}{30}$ $\implies u = \frac{1}{1080}$	**Equations for CLPR solver**

Fig. 17 Calculation of $g(H)$.

invented relations (such as abc4_1, abc4_1_1, ..) introduced during Meta-Interpretation (Figure 14).

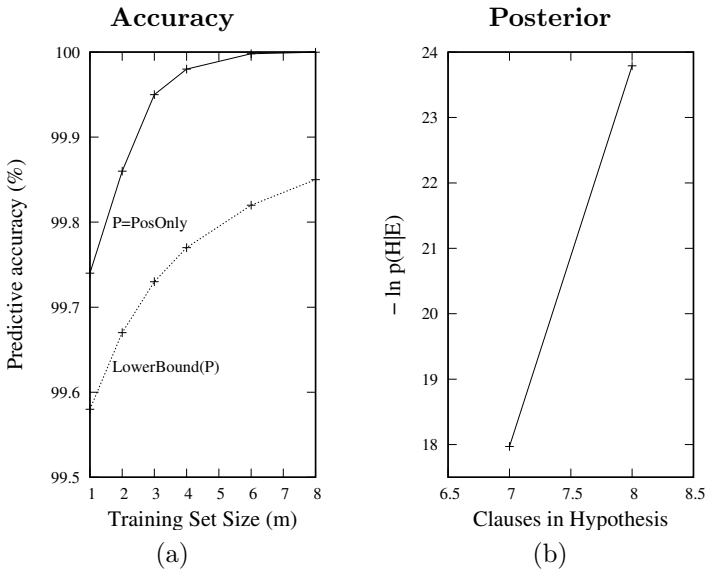


Fig. 18 Problem1: a) Accuracy increase and b) negative log probability decrease.

Training time taken on a laptop (Intel-i7/2.80GHz) is 0.15s for Meta-Compilation and 0.56s for Meta-Interpretation. The generality of the hypothesised program is $g(H) \approx 0.0009$, leading to an Expected-Accuracy of $EA(1,T) > 99.57\%$ and negative log posterior of 17.97. This corresponds to the low-generality criterion for one-shot learning, shown in Figure 11. Figure 17 illustrates the way in which the value of $g(H)$ is derived from equations using SWI-Prolog's CLPR rational solver [9]. CLPR solves rational number equational constraints based on a polynomial-time solver. The

Target	**Example** e^+	**Primitives** P
fire16	at(8,8,16) → at(10,10,1)	Library 62 primitives

Minimal certificates	$\{\langle ws, ss, es, ns, ws, ss, d, d, \ldots, d, d, ns, es\rangle\}$

Hypothesised Program H (7 clauses)	**Training**
fire16(X,Y) :- ws(X,Z), fire16_1(Z,Y). fire16_1(X,Y) :- ss(X,Z), fire16_1_1(Z,Y). fire16_1_1(X,Y) :- ns(X,Z), es(Z,Y). fire16_1_1(X,Y) :- d(X,Z), fire16_1_1(Z,Y). fire16_1_1(X,Y) :- es(X,Z), fire16_1_1_1(Z,Y). fire16_1_1_1(X,Y) :- ns(X,Z), fire16(Z,Y).	$Time = 0.2s/4.14s$ $g(H) = \frac{1}{1079} \approx 0.0009$ $-ln\ p(H\|E) = 32.57$ $EA(1,T) > 99.57\%$

Fig. 19 Problem2: Example, hypothesis and training. The example indicates that the agent starts at coordinate $\langle 8, 8, 16\rangle$ position **A** (see Figure **??**) on floor 16 and needs to reach coordinate $\langle 10, 10, 1\rangle$, position **Exit** on Floor 1. Primitives *ws, ss, ns* and *es* repeatedly move the agent in a given direction until obstructed, while primitive d moves the agent down one floor. While the Minimal Certificate represents an optimal plan for getting to the **Exit** from **A** on floor 16, the hypothesised program will get the agent to the **Exit** from **A**, **B** or **C** from any floor of the building.

equations used in Figure 17 are derived by Deeplog from the definitions in the hypothesis. For instance, $w = \frac{1}{6}^2 + \frac{w}{6}$ is the sum of the generalities of the first and second clause of $abc4\_1\_1$[17]. The generality of the first clause is $\frac{1}{6}^2$ since $g(g) = g(h) = \frac{1}{6}$ and g, h are selected from the 6 non-invented relations in the identified program. The generality of the second clause is $\frac{w}{6}$ since it is the product of $g(abc\_1\_1)$ and $g(cdef)$.

Figure 18 shows a) a comparison of Actual predictive accuracy for DeepLog versus the positive-only Expected-Error bound of Figure 10 and b) the variation of negative log posterior with decreasing clause bounds.

Problem2: Fire escape plan

This problem involves a general fire escape plan for leaving a 16 storey building. The floorplan is shown in Figure 22. All floors have the same layout as Floor 16, except Floor 1, which contains the EXIT.

Figures 19 and 20 show the problem description and results for the fire escape problem. Figure 20a shows once more that the learned program has low generality leading to high predictive accuracy from the first example provided, in accordance with the predictions of the Positive-only Expected-Error bound in Figure 10. Additionally, Figure 20b indicates how two cycles of the Deeplog's binary search process (see Figure 15) successively reduce the number of clauses and negative log posterior of the hypothesis.

[17] The recursive definition of $abc4\_1\_1$ is reflected in the infinite sum $w = (\frac{1}{6}^2 + \frac{w}{6}) = (\frac{1}{6}^2 + \frac{1}{6}^3 + \frac{1}{6}^4 + \ldots)$. CLPR efficiently solves sets of such interrelated equations to provide an exact rational solution.

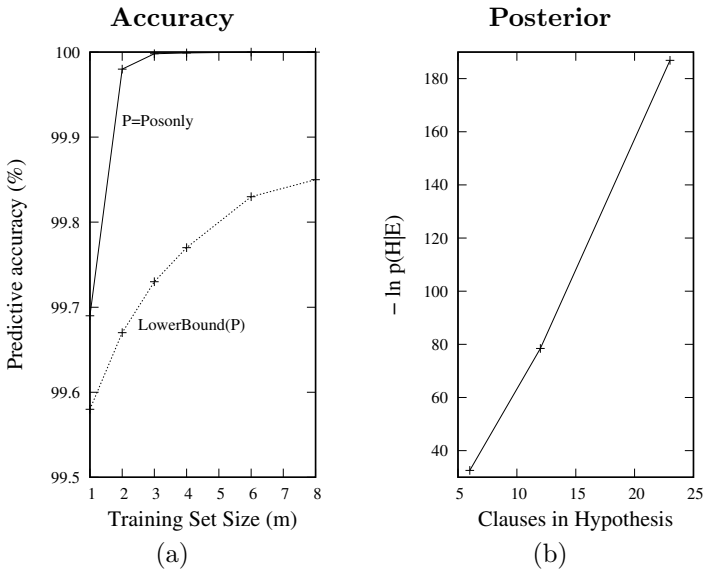


Fig. 20 Problem2: a) Accuracy increase and b) negative log probability decrease.

Target	**Example** e^+	**Primitives** P
rvu	⟨a,l,i,c,e⟩ →⟨E,C,I,L,A⟩	Library 62 primitives

Minimal certificates	$\{\langle call1, pop, up, psh, pop, up, psh, pop, up, psh, pop, up, psh, pop, up, psh, ret1\rangle\}$

Output Hypothesis H (5 clauses)	**Training**
rvu(X,Y) :- call1(X,Z), rvu_1(Z,Y).	$Time = 0.21s/0.65s$
rvu_1(X,Y) :- pop(X,Z), rvu_1_1(Z,Y).	$g(H) = \frac{1}{7740} \approx 0.0001$
rvu_1_1(X,Y) :- up(X,Z), rvu_1_1_1(Z,Y).	
rvu_1_1_1(X,Y) :- psh(X,Z), ret1(Z,Y).	
rvu_1_1_1(X,Y) :- psh(X,Z), rvu_1(Z,Y).	$EA(1,T) > 99.94\%$

Fig. 21 Problem3: Example, hypothesis and training. The primitive *call1* pushes an empty list *A* onto the program stack as an Accumulator. Primitive *pop* removes the head of the input list and pushes it onto the program stack. Primitive *psh* replaces the top two elements *X,Z* of the program stack by a list with head *X* and tail *Y*. Primitive *up* replaces the letter at the top of the program stack by its uppercase version. Lastly, *ret1* replaces the program stack *Y|Ident* by the output list *Y*.

Problem3: Reverse uppercase

We now consider the "alice" to "ECILA" textual analogy problem. This is shown in Figure 21, which involves the construction of a function for reversing and making uppercase a letter sequence. Efficient Prolog programs normally require the introduction of an arity 3 auxiliary predicate to reverse a list. The same end is achieved with arity 2 predicates for DeepLog by dynamically creating a stack in the program state. Since Problem3 requires such a stack to be of unbounded size, the resulting

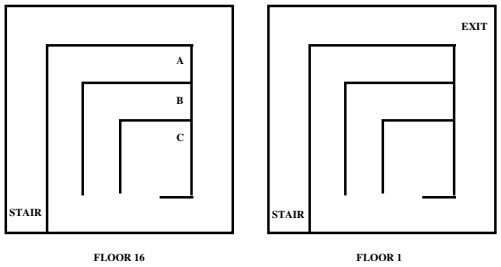


Fig. 22 Problem2: Floorplan.

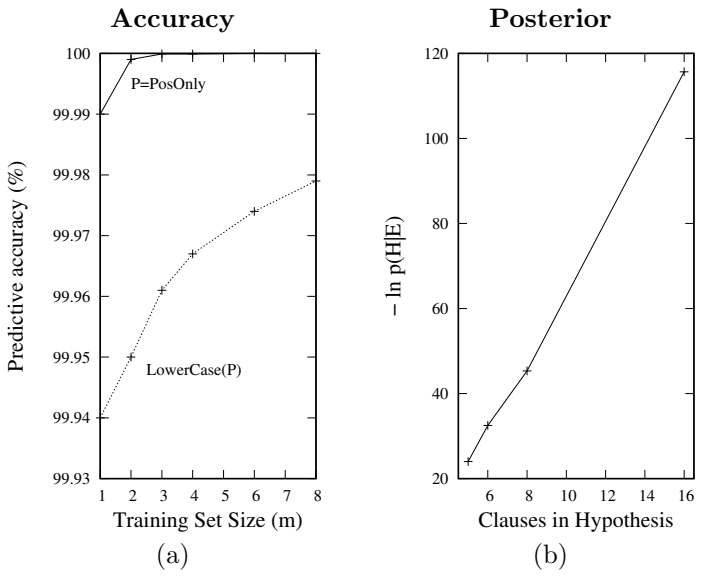


Fig. 23 Problem3: a) Accuracy increase and b) negative log probability decrease.

program can be thought of as a Push-Down Automaton (PDA) as opposed to the Finite State Automata (FSA) solutions in Problem1 and Problem2. PDAs represent a higher expressivity function class than FSAs since the stack can be treated as an unbounded Turing-machine tape. This additional functionality is enabled by the availability of the background library primitives *call1* (see Figure 21), which introduces a new "empty" object on the top of the calling stack. The type of the new object, in this case a list, is assigned by DeepLog using type-recognition predicates applied to the example provided. On exit this object is passed back using *ret1*. It should be noted that the resulting program would be relatively complex to encode manually, since it involves introduction of three subsidiary relations with mutual recursion and efficiently interleaving of pushing, popping and uppercasing of letters. Training is achieved with Expected-Accuracy from one example in under 1s on a laptop, and the generality and negative log probability are lower than in Problem1 and Problem2. This is reflected by

more rapid accuracy and posterior convergence in Figure 23a. Figure 23b shows that posterior converges is achieved in three binary search iterations (see Figure 15) which reduce the hypothesis from 16 clauses down to the final 5 clause hypothesis returned.

Problem	**Clauses DL**	**Clauses RDL**	**EA(%)**	**Time(s) DL**	**Time(s) RDL**	**Time Ratio**
abcs04	7	9	99.6	0.71	0.043	16.5
abcs08	11	14	99.1	1.12	0.076	14.7
abcs12	15	17	99.8	11.54	0.142	81.3
ancestor	2	3	38.1	0.06	0.029	1.7
descendant	2	3	38.1	0.05	0.029	1.7
bigben04	>37	29	100.0	>26.1	0.053	>491.0
bigben08	>37	29	100.0	>26.1	0.055	>474.5
bigben12	>37	29	100.0	>26.1	0.056	>466.1
fire04	18	17	99.6	1.22	0.030	40.7
fire08	18	17	99.6	1.27	0.048	26.5
fire16	18	17	99.6	1.96	0.050	39.2
fire32	18	17	99.6	4.34	0.051	81.9
fire64	-	17	99.6	-	0.059	-
fire128	-	17	99.6	-	0.060	-
member04	2	3	94.25	0.047	0.046	1.0
member08	2	3	91.4	0.066	0.049	1.0
re1	4	3	91.4	0.082	0.045	1.8
re2	3	3	90.3	0.10	0.036	2.8
revupc05	5	6	99.9	0.69	0.058	13.0
revupc09	5	6	99.9	8.87	0.143	62.0
sumlist03	3	4	90.3	0.08	0.049	1.6
sumlist06	3	4	90.3	0.82	0.063	13.0
sumlist09	3	4	90.3	9.74	0.193	50.5
sumlist12	-	4	90.3	-	0.799	-

Fig. 24 One-shot learning comparison of **DeepLog (DL)** and ReDuceDL (RDL) on Clauses (DL vs RDL), Expected Accuracy (EA) is defined as EA(1,T) = 100(1-EE(H)), Learning Time (DL vs RDL) and Time Ratio (DL/RDL).

5.2 Experimental Comparison of ReDuceDL vs DeepLog

We now compare a superset of the DeepLog results presented in Subsection 5.1 to a range of ReDuceDL experiments in Figure 24. As before, each target program uses the same background library of 62 primitive relations. The library contains a set of type identifiers and basic relations over various types of entities, including characters, lists, stacks, numbers, 3D-positions, chess and family relations. The Problems are into groups (eg. fire), which vary according to the size of input (eg. fire04,. . . fire128).

We consider the following questions based on the outcome of the experiments. [18]

Question 5.1: Does ReDuceDL outperform DeepLog, in terms of time efficiency, on the initial DeepLog problems (abcs, fire, revupc)?
Answer 5.1: Yes, in all tested variants ReDuceDL outperform DeepLog in terms of time efficiency with a time ratio of **abcs** (16.5-81.3), **fire** (40.7-81.9) and **revupc** (13.0-62.0).

Question 5.2: Does ReDuceDL outperform DeepLog, in terms of time efficiency, on problems other than the initial DeepLog problems (ancestor, bigben, member, re, sumlist)?
Answer 5.2: Yes, in all tested variants other than member ReDuceDL outperforms DeepLog in terms of time efficiency with a time ratio of ancestor (1.7), bigben (466.1-491.0), fire (40.7-81.9), member (1.0), re (1.8-2.8) and sumlist (1.6-50.5).

6 Conclusions and Further Work

6.1 Conclusions

This paper describes a new Inductive Logic Programming system, ReDuceDL, which integrates the author's previous DeepLog and ReDuce systems. In Section 1 motivating examples are provided involving the learning of a) Prolog programs which represent two Recursively Enuerable (RE) sets and b) a strategy to allow optimally efficient fire escape routes from bedrooms in a high-rise building. However, owing to non-linear time and space complexity, DeepLog is unable to find a correct program for building designs higher than 16 storeys. By contrast, the new ReDuceDL system outperforms DeepLog on this problem by a factor of more than 80, and easily scales to at least 128 storeys due to the incorporation of ReDuce's greedy folding approach.

6.1.1 Inputs and Outputs

The ReDuceDL experiments and implementation in this paper assume examples take the form of Input/Output pairs such as *revupc([a,b,c],['C','B','A'])*. ReDuceDL first calls DeepLog to find a minimal length sequence of primitives which transform the Input (eg *[a,b,c]*) to the Output (*['C','B','A']*). The minimal certificate is ⟨*call1,pop,upcase,push,pop,upcase,push, pop,upcase,push,return1*⟩. This certificate is passed to ReDuce which uses greedy folding to find a compact hypothesis.

Section 2 compares the Meta-Interpretive hypothesis generation approach of DeepLog to the grammar-based compression approach employed by ReDuceDL. The ReDuceDL framework is introduced in Section 3.6, which provides and contrasts definitions of grammar-based *lossless compression* versus a new form of grammar-based *inductive compression*. Lossless and Inductive compression algorithms are shown to terminate and a simple example is provided to contrast the outcome of greedy lossless compression versus greedy inductive compression.

[18] Timings in Figure 24 for DeepLog and ReDuce were both based on SWI Prolog runnning on a single core 11th Gen Intel(R) i7-1165G7 @ 2.80GHz.

The ReDuce algorithm is described in Section 4.2. Both the correctness and linear-time complexity are shown. In Section 5.2 the three problems from [18], used to exemplify DeepLog, are revisited. In all cases ReDuce is at least 1,000 times faster in hypothesising an equivalent Prolog program to that found by DeepLog. Moreover, in Problems 1 and 2 ReDuce introduced predicates which were provided to DeepLog either as user-defined background knowledge or by special purpose mechanisms.

6.2 Further Work

6.2.1 Bayes' Model versus Smallest Grammar

The DeepLog paper [18] was motivated by demonstrating that programs can be learned from a small number of positive-only examples. The Bayes' model developed in that paper showed that learning can be achieved, with high probability, from a single example in the case that both a) the generality and b) the size of the target program are low. Problems 1, 2 and 3 in Section 3.6 have these properties.

The implementation of Algorithm 1 in Section 4.2 is presented alongside formal proofs of Correctness and Time Complexity. Further formal optimality analysis along the lines of [2] would be required to identify how close learned programs are to a smallest grammar. However, when learning from more than one example, any such analysis should be combined with sample complexity bounds related to those associated with DeepLog [18]. The Bayes' model bounds in DeepLog indicate that error is not minimised by identifying the smallest grammar H, but rather by minimising $sz(H) + m\,(ln\ g(H))$, where $sz(H) = -ln(D_{\mathcal{H}}(H))$ represents the size of the hypothesis. $g(H)$ is the generality of H where $0 \leq g(H) \leq 1$ and $g(H)$ is the sum of the probability of the instances of H.

Further work is required to show how close the greedy approach is to learning a Bayes' predictor with minimal expected error.

6.2.2 Expressivity of learned programs

Section 5.1 describes experiments on three problems. Problem 1 (abc4), Problem 2 (fire16) and Problem 3 (revupc) can each be viewed either as learning a finite automaton or a Chomsky Type-3 regular grammars based on the sequence of the primitive actions provided. However, Problems 1 and 2 simply involve sequentially reading the input tape, while Problem 3 involves a) reading the input tape, b) creating and operating an internal tape representing a stack and c) writing to the output tape. In this way, Problem 3 can be most easily characterised as either a pushdown automaton or a Universal Turing machine program.

Further work is required to identify primitives, other than stack operations on an unbounded tape, which might support learning Type-0 grammars, equivalent to Recursively Enumerable functions and Universal Turing Machine programs.

6.2.3 Identification of noise

It should be noted that all the learned programs in the paper represent surjective functions. In this case, greedy inductive compression of two examples of the same

surjective function leads to the same hypothesis. This is demonstrated in the following cases, in which three examples are represented with distinct separators, *sep1* and *sep2*, which are presented to ReDuce within the same example.

Example set	$S \rightarrow \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, sep1, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, sep2, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}$
ReDuce	$S \rightarrow \mathbf{Star1}, sep1, \mathbf{Star1}, sep2, \mathbf{Star1}$

The ReDuce compressed sequence above show that the three example sequences are members of the same function,**Star1**. The approach appears to identify a *Normal Form* hypothesis (in this case **Star1**) by learning from a set of several examples. If this can be formally demonstrated it would avoid the potentially high costs associated with deductive testing of examples, since, by construction, each example generalises to the same hypothesis. Furthermore, negative examples and noisy data could be identified using the same mechanism as follows.

Example set	$S \rightarrow \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, sep1, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{e}, sep2, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}$
ReDuce	$S \rightarrow \mathbf{Star1}, sep1, \mathbf{Star1}, \mathbf{c}, \mathbf{e}, sep2, \mathbf{Star1}$

In this case **Star1,c,e** represents either an error, a negative example or an exception to the general rule.

In further work the author aims to formally identify whether, when provided with multiple example sequences, Algorithm 1 always generates a unique hypothesis when provided with noise-free positive examples, allowing noisy examples to be identified.

6.2.4 Large-scale testing

In addition to further theoretical work on the identification of noise described above, further work is required to extend the benchmarks from the DeepLog paper. In particular, the author aims to test a broad range of tasks, including learning larger-scale programs as well as noisy real-world datasets. Error estimates will be introduced to better understand robustness and noise. Additionally the effects of scaling to large scale problems will be investigated.

6.2.5 Learning disjunctive concepts

In further work the author aims to develop inductive compression with further underlying theory, implementations and experiments. We also intend to develop the representation and theory to address issues related to learning of arbitrary Prolog relations, such as *member* and *ancestor* from small numbers of examples.

6.2.6 Learning 2D grammars and images

Lastly, the author hopes to explore inductive compression approaches for learning 2D grammars, as an approach to learning from images.

Acknowledgements

The author acknowledges support from both Nanjing University and the UK's EPSRC Human-Like Computing Network (EP/R022291/1), for which he acted as director. The author would also like to acknowledge helpful discussions on this topic with both Dr Lun Ai and Dr Wang-Zhou Dai. Lastly the author would like to thank his wife, Thirza, for her unfailing encouragement and willingness to always listen with care during the development of the work described.

References

[1] Anthony, M.H., Biggs, N.: Computational learning theory (1997)
[2] Casel, K., Fernau, H., Gaspers, S., Gras, B., Schmid, M.: On the complexity of the smallest grammar problem over fixed alphabets. Theory of Computing Systems **65**, 344–409 (2021), https://doi.org/10.1007/s00224-020-10013-w
[3] Church, A.: An unsolvable problem of elementary number theory. American Journal of Mathematics **58**, 345–363 (1936)
[4] Cropper, A., Dumančić, S., Evans, R., Muggleton, S.: Inductive Logic Programming at 30. Machine Learning **111**, 147–172 (2021), http://www.doc.ic.ac.uk/~shm/Papers/ilp30.pdf
[5] De Raedt, L.: Logical seetings for concept learning. Artificial Intelligence **95**, 187–201 (1997)
[6] Gold, E.: Language identification in the limit. Information and Control **10**, 447–474 (1967)
[7] Gold, E.: Complexity of automaton identification from given data. Information and Control **37**, 302–320 (1978)
[8] Goulden, R., Nation, P., Read, J.: How large can a receptive vocabulary be? Applied linguistics **11**(4), 341–363 (1990)
[9] Harvey, W., Stuckey, P.: A unit two variable per inequality integer constraint solver for constraint logic programming. University of Melbourne (1995)
[10] Huet, G.: A unification algorithm for typed λ-calculus. Theoretical Computer Science **1**(1), 27–57 (1975)
[11] Kearns, M.J., Vazirani, U.V.: Computational learning theory. ACM SIGACT News **26**(1), 43–45 (1995)
[12] Lake, B., Salakhutdinov, R., Gross, J., Tenenbaum, J.: One shot learning of simple visual concepts. In: Proceedings of the 33rd Annual Conference of the Cognitive Science Society. pp. 2568–2573 (2011)
[13] Li, F.F., Fergus, R., Perona, P.: One-shot learning of object categories. IEEE transactions on pattern analysis and machine intelligence **28**(4), 594–611 (2006)
[14] McNeill, D.: Developmental psycholinguistics. In: Miller, G., McNeill, D. (eds.) Handbook of Social Psychology (1966)
[15] Muggleton, S.: Duce, an oracle based approach to constructive induction. In: IJCAI-87. pp. 287–292. Kaufmann (1987), http://www.doc.ic.ac.uk/~shm/Papers/ijcai87.pdf
[16] Muggleton, S.: Inductive Logic Programming. New Generation Computing **8**(4), 295–318 (1991), http://www.doc.ic.ac.uk/~shm/Papers/ilp.pdf

[17] Muggleton, S.: Learning from positive data. In: Muggleton, S. (ed.) Proceedings of the Sixth International Workshop on Inductive Logic Programming (Workshop-96). pp. 358–376. LNAI 1314, Springer-Verlag, Berlin (1996), http://www.doc.ic.ac.uk/~shm/Papers/poslearn.pdf

[18] Muggleton, S.: Hypothesising an algorithm from one example: the role of specificity. Philosophical Transaction of the Royal Society A **381:20220046** (2023), https://royalsocietypublishing.org/doi/10.1098/rsta.2022.0046

[19] Muggleton, S.: Reduce: Linear-time inductive compression using greedy folding. In: Proceedings of the 5th International Joint Conference on Learning and Reasoning, IJCLR 2025 (2025), http://www.doc.ic.ac.uk/~shm/Papers/Reduce.pdf, in Press

[20] Muggleton, S., Lin, D., Pahlavi, N., Tamaddoni-Nezhad, A.: Meta-interpretive learning: application to grammatical inference. Machine Learning **94**, 25–49 (2014). https://doi.org/10.1007/s10994-013-5358-3, https://link.springer.com/article/10.1007/s10994-013-5358-3

[21] Muggleton, S., Lin, D., Tamaddoni-Nezhad, A.: Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. Machine Learning **100**(1), 49–73 (2015), https://link.springer.com/article/10.1007/s10994-014-5471-y

[22] Muggleton, S., Raedt, L.D.: Inductive logic programming: Theory and methods. Journal of Logic Programming **19,20**, 629–679 (1994), http://www.doc.ic.ac.uk/~shm/Papers/lpj.pdf

[23] Nevill-Manning, C., Witten, I.: Identifying hierarchical structure in sequences: A linear-time algorithm. Journal of Artificial Intelligence Research **7**, 67–82 (1997), https://arxiv.org/pdf/cs/9709102

[24] Patsantzis, S., Muggleton, S.: Meta-interpretive learning as metarule specialisation. Machine Learning **111**, 3703–3731 (2022), https://link.springer.com/article/10.1007/s10994-022-06156-1

[25] Rice, H.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society **74**(2), 358–366 (1953)

[26] Shaban, A., Bansal, S., Liu, Z., Essa, I., Boots, B.: One-shot learning for semantic segmentation. arXiv preprint arXiv:1709.03410 (2017)

[27] Tärnlund, S.A.: Horn clause computability. BIT Numerical Mathematics **17**(2), 215–226 (1977)

[28] Tenenbaum, J.: A Bayesian Framework for Concept Learning. Ph.D. thesis, Massachusetts Institute of Technology (1999)

[29] Turing, A.: On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society **42**, 230–265 (1936)

[30] Valiant, L.: A theory of the learnable. Communications of the ACM **27**, 1134–1142 (1984)

[31] Vinyals, O., Blundell, C., Lillicrap, T., Wierstra, D., et al.: Matching networks for one shot learning. Advances in neural information processing systems **29** (2016)

[32] Zhou, Z.H.: Abductive learning: towards bridging machine learning and logical reasoning. Science China Information Sciences **62**(7) (2019)