Incremental and Iterative Learning of Answer Set Programs from Mutually Distinct Examples

Arindam Mitra and Chitta Baral
Arizona State University
(*e-mail:* {amitra7,chitta}@asu.edu)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Over the years the Artificial Intelligence (AI) community has produced several datasets which have given the machine learning algorithms the opportunity to learn various skills across various domains. However, a subclass of these machine learning algorithms that aimed at learning logic programs, namely the Inductive Logic Programming algorithms, have often failed at the task due to the vastness of these datasets. This has impacted the usability of knowledge representation and reasoning techniques in the development of AI systems. In this research, we try to address this scalability issue for the algorithms that learn answer set programs. We present a sound and complete algorithm which takes the input in a slightly different manner and performs an efficient and more user controlled search for a solution. We show via experiments that our algorithm can learn from two popular datasets from machine learning community, namely bAbl (a question answering dataset) and MNIST (a dataset for handwritten digit recognition), which to the best of our knowledge was not previously possible. The system is publicly available at https://goo.gl/KdWAcV.

KEYWORDS: Inductive Logic Programming, Answer Set Programming, Question Answering, Handwritten Digit Recognition, Context Dependent Learning.

1 Introduction

Answer Set Programming has emerged as a powerful tool for knowledge representation and reasoning. To use this tool for an application, however, one needs application specific knowledge. For E.g., if a system uses answer set programming to answer the question from column 1 in Table 1 the system needs to know that "X is to the right of Y IF Y is to the left of Z and Z is above X". Inductive Logic Programming algorithms aim to learn these kinds of knowledge from a dataset. However, existing ILP algorithms have limited scalabilty and often fail to learn knowledge from a machine learning dataset. This leads to manual construction of a knowledge base which can be very time consuming and may not be practical sometimes. For E.g., for applications where an effective representation of the rules is unknown, such as for the case of handwritten digit recognition (Fig. 1), one may need to try several representations before settling down for a winner. However, this may be unrealistic given that MNIST dataset (Fig. 1) contains 50,000 examples and writing down the rules that explain all these examples for a particular choice of representation will take significant amount of time.

In this work, we consider this scalability issue. We observe that one major obstruction in scalability arises from the discrepancy between the definition of Inductive Logic Programming and the

x	The square is above the rectangle. The triangle is to the left of the square. Is the rectangle to the right of the triangle?	The square is below the rectangle. The triangle is to the right of the square. Is the rectangle to the right of the triangle?	The square is below the rectangle. The triangle is to the right of the square. Is the triangle below the rectangle?
y	Yes	No	Yes

Table 1: A set of examples taken from the Task 17 of bAbI question answering dataset.

Fig. 1: A set of images from the MNIST dataset.

structure of a machine learning dataset. The learning problem in Inductive Logic Programming (ILP) is defined as follows (Muggleton 1991):

Definition 1 (***Inductive Logic Programming***)
Given a set of positive examples E^+, negative examples E^- and some background knowledge B, an ILP algorithm finds an Hypothesis H such that,

$$B \cup H \models E^+, B \cup H \not\models E^-$$

The hypothesis space is restricted with a language bias that is specified by a series of mode declarations M.

A machine learning dataset on the other hand contains a series of $\langle x, y \rangle$ pairs, x being the input and y being the desired output (Table 1). To work with an ILP algorithm, one needs to first convert the $\langle x, y \rangle$ pairs in the format of $\langle B, E^+, E^- \rangle$. The conversion process is carried out by the user and so there might be some variations. However, normally the sets E^+ and E^- are created using y's and the x's go inside B. Extra care is taken so that different $\langle x, y \rangle$ pairs do not interfere with each other. Table 2(a) shows one example of this process. Since the number of $\langle x, y \rangle$ pairs are usually large, the problem instance becomes too big for the ILP solvers to handle . For example, consider someone wants to employ an ILP algorithm to learn from a question answering task from bAbI dataset (Weston et al. 2015), which contains $1,000$ comprehension examples similar to the ones in Table 1. The resulting background knowledge B will contain about $10,000$ facts and E^+ will contain $1,000$ positive annotations pertaining to answers and E^- will contain a total of $1,000$ negative examples describing what is not an answer for each question. An ILP solver such as XHAIL (Ray 2009) will throw memory errors when given an input of this size. The question that we ask here is "can we find a solution to the ILP problem without considering all the $\langle x, y \rangle$ pairs together ?" We show that the answer is yes. In fact it is possible to find a solution considering only one $\langle x, y \rangle$ pair at a time. To achieve this we model the learning task as follows:

Definition 2 (***Inductive Logic Programming for Distinct Examples***)
An ILP task for *Distinct Examples* (denoted as ILP^{DE}) is a tuple $\langle B, M, D \rangle$, where B is an Answer Set Program, called the background knowledge, M defines the set of rules allowed in hypotheses

(the hypothesis space) and D is the dataset containing a series of context dependent examples $\langle E_1, E_2, ..., E_n \rangle$. Here each E_i is a tuple $\langle O_i, E_i^+, E_i^- \rangle$ where, O_i is a logic program, called *observation* , E^+ is a set of positive ground literals and E^- is a set of negative ground literals. A hypothesis H is an inductive solution of T (written as $H \in ILP^{DE}(B, M, D)$) *iff*,

$$H \cup B \cup O_i \vdash E_i^+, \ \forall i = 1...n$$

$$H \cup B \cup O_i \nvdash E_i^-, \ \forall i = 1...n$$

In this formulation, each example $\langle O_i, E_i^+, E_i^- \rangle$ directly corresponds to an $\langle x, y \rangle$ pair and it takes into consideration that there are several distinct examples in a dataset, so there is no need to explicitly isolate them from each other. Table 2(b) shows the encoding of the running example in the format of ILP^{DE}. It turns out that the ILP^{DE} task described here is a simplification of the *Context-dependent Learning from Ordered Answer Sets task* proposed in (Law et al. 2016). However, to solve the *Context-dependent Learning from Ordered Answer Sets task* the authors in (Law et al. 2016) convert it to a standard ILP problem which creates the same scalability issue.

Table 2: The *sample* predicate is used to separate different examples. The constants tri, rec, sq respectively denote triangle, rectangle and square. $holdsAt(rp(sq, rec, above), 1)$ says that the square is above the rectangle at time point 1.

	$ans(X, no) \leftarrow not\ ans(X, yes), id(X).$
B	$sample(1, holdsAt(rp(sq, rec, above), 1)).$ $sample(1, holdsAt(rp(tri, sq, left), 1)).$ $ans(1, yes) \leftarrow$ $\quad sample(1, holdsAt(rp(rec, tri, right), 1)).$
	$sample(2, holdsAt(rp(sq, rec, below), 1)).$ $sample(2, holdsAt(rp(tri, sq, right), 1)).$ $ans2(yes) \leftarrow$ $\quad sample(2, holdsAt(rp(rec, tri, right), 1)).$
	$sample(3, holdsAt(rp(tri, sq, left), 1)).$ $sample(3, holdsAt(rp(tri, sq, left), 1)).$ $ans(3, yes) \leftarrow$ $\quad sample(3, holdsAt(rp(tri, rec, below), 1)).$
E^+	$\{ans(1, yes), ans(2, no), ans(3, yes).\}$
E^-	$\{ans(1, no), ans(2, yes), ans(3, no).\}$

(a) An ILP encoding of the problem in Table 1

E_1	O_1	$holdsAt(rp(sq, rec, above), 1).$ $holdsAt(rp(tri, sq, left), 1).$ $ans(yes) \leftarrow holdsAt(rp(rec, tri, right), 1).$
	E_1^+	$\{ans(yes)\}$
	E_1^-	$\{ans(no)\}$
E_2	O_2	$holdsAt(rp(sq, rec, below), 1).$ $holdsAt(rp(tri, sq, right), 1).$ $ans(yes) \leftarrow holdsAt(rp(rec, tri, right), 1).$
	E_2^+	$\{ans(no)\}$
	E_2^-	$\{ans(yes)\}$
E_3	O_3	$holdsAt(rp(tri, sq, left), 1).$ $holdsAt(rp(tri, sq, left), 1).$ $ans(yes) \leftarrow holdsAt(rp(tri, rec, below), 1).$
	E_3^+	$\{ans(yes)\}$
	E_3^-	$\{ans(no)\}$

(b) An ILP^{DE} encoding of the problem in Table 1

It should be noted that any standard ILP problem $\langle B, M, E^+, E^- \rangle$ can be thought of as an ILP^{DE} problem with only one example, $\langle \{\}, M, \langle (B, E^+, E^-) \rangle \rangle$. Similarly any ILP^{DE} task can be converted to an ILP task. However, utilizing the 'distinctness' property of the examples we can do better. The algorithm that we propose here roughly works as follows: Given an instance of the ILP^{DE} task, it first finds a solution H_1 of E_1. Then it expands H_1 minimally to solve only E_2 and obtains H_2 . In the next iteration it again expands H_2 minimally to solve E_1 and it continues expanding until it finds a hypothesis that solves both E_1 and E_2. Next it starts with a solution of $\langle E_1, E_2 \rangle$ and tries to expand it iteratively until it solves all of E_1, E_2 and E_3. The process continues until a hypothesis is found that explains all the examples. Section 3 describes the algorithm. We show that the algorithm is sound and complete when $H \cup B \cup O_i$ is *stratified* for all $i = 1, ..., n$.

Our algorithm allows more control over the mode declarations (Section 2) which can lead to noticeable speed up in the search process. We evaluate our algorithm on two popular datasets: 1) a question answering dataset published by Facebook AI Research (Weston et al. 2015) and 2) a handwritten digit recognition database (LeCun 1998). To the best of our knowledge, no sound and complete ILP algorithm could learn from these two datasets. The work of (Mitra and Baral 2016) that learns from the bAbl dataset uses a modification of an existing ILP algorithm and the resulting algorithm is not complete. We discuss this further in section 5.

2 Background

In this section, we describe the type of rules that our algorithm can deal with, the syntax of the mode declarations and the XHAIL algorithm which plays a crucial role in our algorithm.

Answer Set Programming

An answer set program is a collection of rules of the form,

$$L_0 \leftarrow L_1, ..., L_m, \textbf{not } L_{m+1}, ..., \textbf{not } L_n$$

where each of the L_i's is a literal in the sense of a classical logic. Intuitively, the above rule means that if $L_1, ..., L_m$ are true and if $L_{m+1}, ..., L_n$ can be safely assumed to be false then L_0 must be true. The left-hand side of an ASP rule is called the *head* and the right-hand side is called the *body*. Predicates and ground terms in a rule start with a lower case letter, while variable terms start with a capital letter. We will follow this convention throughout the paper. A rule with no *head* is called a *constraint*. A rule with empty *body* is referred to as a $fact$. The semantics of ASP is based on the stable model semantics of logic programming (Gelfond and Lifschitz 1988). In this work, both the background knowledge B and the solution H are a collection of such ASP rules.

Mode Declarations

Given a set of positive examples E^+, negative examples E^- and some background knowledge B, an ILP algorithm computes a set of rules H so that $B \cup H \models E$. The rules in H are often restricted with a language bias that is specified by a series of mode declarations M (Muggleton 1995). One can think of this as a way of injecting expert knowledge for the learning task.

There are two types of mode declarations, namely *modeh* declarations and *modeb* declarations. A *modeh(s)* declaration (Table 3) specifies a literal s that can appear as the head of a rule in H. A *modeb(s)* declaration (Table 3) specifies a literal s that can appear in the body of a rule. The argument s is called *schema* and comprises of two parts: 1) an *identifier* for the literal and 2) a list of *placemakers* for each argument of that literal. A *placemaker* is either *+type* (input), *-type* (output) or *$type* (constant), where *type* denotes the type of the argument. An answer set rule is in the hypothesis space defined by M (call it $L(M)$) if and only if its head (resp. each of its body literals) is constructed from the schema s in a *modeh(s)* (resp. in a *modeb(s)*) in $L(M)$) as follows:

- by replacing an output (-) placemaker by a new variable.
- by replacing an input (+) placemaker by a variable that appears in the head or in a previous body literal and
- by replacing a ground ($) placemaker by a ground term.

Table 3 shows a set of mode declarations M_{sample} that one can use to solve the example problem in Table 1.There is only one *modeh(s)* declaration in M_{sample}, where the schema is *holdsAt(relativeposition(+op1,+op1, $direction), +time)*. Assuming that there are only four constants of type *directions*, the set of possible head literals are:

$$\begin{Bmatrix} holdsAt(relativeposition(X,Y,left),T), \\ holdsAt(relativeposition(X,Y,right),T), \\ holdsAt(relativeposition(X,Y,above),T), \\ holdsAt(relativeposition(X,Y,below),T) \end{Bmatrix}$$

Where X and Y are variables of type $op1$ and T has type *time*. There are three *modeb* declarations and they restrict additions of literals to the body as directed by their individual schema. Note that the following rule,

$$holdsAt(relativeposition(X,Y,left),T) \leftarrow holdsAt(relativeposition(Z,X,above),T), \\ holdsAt(relativeposition(Y,Z,right),T).$$

is in $L(M_{sample})$, as the head is allowed by the *modeh* (Table 3) and the third *modeb* (Table 3) allows the addition of $holdsAt(relativeposition(Z,X,above),T)$ with Z being an output (new) variable and the first *modeb* allows the addition of $holdsAt(relativeposition(Y,Z,right),T)$, as all the associated variables Y, Z and T have appeared before.

#modeh holdsAt(relativeposition(+op1,+op1,$direction),+time).
#modeb holdsAt(relativeposition(+op1,+op1,$direction),+time).
#modeb holdsAt(relativeposition(+op1,-op1,$direction),+time).
#modeb holdsAt(relativeposition(-op1,+op1,$direction),+time).

Table 3: Mode declarations for the problem of Table 1

Additionally, *weights* can be assigned to *modeh* and *modeb* (written as *#modeh(s)=W*) and they express the cost that is involved when a mode declaration is used. The default weight for mode declarations is 1. Existing implementations of the ILP algorithms, take only one set of mode declarations and thus all the *modeh* declarations share the same set of *modebs*. Our algorithm allows the user to provide *modeh* specific *modeb* declarations. This additional feature allows the user to provide more supervision in the search procedure and makes the search faster.

XHAIL

The XHAIL (Ray 2009) algorithm plays a crucial role in the algorithm that we present here. In this section, we describe various concepts and notations associated with the XHAIL algorithm. Given an ILP task $ILP(B,M,E=\{E^+ \cup E^-\})$, XHAIL (Ray 2009) derives the hypothesis in three steps, namely the *abductive* step, the *deductive* step and the *inductive* step. We will explain these steps with respect to the example E_1 from Table 2(b). The set B contains the representation of x_1, denoted by O_1 and the set E the contains annotations derived from y_1. M is the set of mode declarations described in Table 3.

Abductive Step

In the first step XHAIL finds a set of ground (variable free) atoms $\triangle = \{\alpha_1, ..., \alpha_n\}$ such that $B \cup \triangle \models E$, where each α_i is a ground instance of the *modeh(s)* declaration atoms. For the running example there is only one *modeh* declaration. Thus the set $\triangle$ can contain ground instances of only $holdsAt(relativeposition(X,Y,Z),T)$. In the following we show one possible $\triangle$ that meets the above requirement.

$$\triangle = \Big\{ holdsAt(relativeposition(rectangle, triangle, right), 1) \quad \Big\}$$

Deductive Step

In the second step, XHAIL computes a clause $\alpha_i \leftarrow \delta_i^1 ... \delta_i^{m_i}$ for each α_i in $\triangle$, where $B \cup \triangle \models \delta_i^j, \forall 1 \leq i \leq n, 1 \leq j \leq m_i$ and each clause $\alpha_i \leftarrow \delta_i^1 ... \delta_i^{m_i}$ is a ground instance of a rule in $L(M)$. In the running example, $\triangle$ contains only one atom, $\alpha_1 = holdsAt(relativeposition(rectangle, triangle, right), 1)$ which is initialized to the head of the clause k_1. The body of k_1 is saturated by adding all possible ground instances of the literals in *modeb(s)* declarations that satisfy the constraints mentioned above. There are two ground instances, $holdsAt(relativeposition(square, rectangle, above), 1)$ and $holdsAt(relativeposition(triangle, square, left), 1)$, of the literals in the *modeb(s)* declarations and both of them can be added to the body as specified by M. In the following we show the set of ground clauses K (called *kernel*) constructed in this step and their variabilized version K_v (called *generalization*) that is obtained by replacing all input and output terms by variables.

$$K = \left\{ \begin{array}{l} holdsAt(relativeposition(rectangle, triangle, right), 1) \\ \quad \leftarrow holdsAt(relativeposition(square, rectangle, above), 1), \\ \quad\quad holdsAt(relativeposition(triangle, square, left), 1). \end{array} \right\}$$

$$K_v = \left\{ \begin{array}{l} holdsAt(relativeposition(X,Y,right), T) \\ \quad \leftarrow holdsAt(relativeposition(Z,X,above), T), \\ \quad\quad holdsAt(relativeposition(Y,Z,left), T). \end{array} \right\}$$

Inductive Step

In this step XHAIL tries to find a compressive theory H by selecting from K_v as few literals as possible while ensuring that $B \cup H \models E$. For this example, working out this problem will lead to a unique solution,

$$H = \Big\{ holdsAt(relativeposition(X,Y,right), T). \quad \Big\}$$

which contains a single rule with empty body. In general, the compression process may lead to multiple options for H.

Let $\langle H_I, H_G, \triangle \rangle$ denote a solution returned by $XHAIL(B, M, E)$, where H_G is the generalization computed from $\triangle$ and H_I is a compressed version of H_G that solves E. It should be noted that there might be many choices for $\triangle$ and correspondingly there might be many possible solutions $\langle H_I, H_G, \triangle \rangle$. In the following table, we define few notations which will be useful later.

Notations	
XHAIL(B,M,E)	The set of all the solutions $\langle H_I, H_G, \triangle\rangle$ to the problem $P = ILP(B, M, E)$, where H_I is minimal i.e. no compressed version of H_I can solve P.
$\triangle(B, M, E)$	$\{\triangle \mid \langle H_I, H_G, \triangle\rangle \in XHAIL(B, M, E)$ for some $H_I, H_G\}$.
$H_G(B, M, E)$	$\{H_G \mid \langle H_I, H_G, \triangle\rangle \in XHAIL(B, M, E)$ for some $\triangle, H_I\}$.
$H_G(\triangle)$	The generalization computed from $\triangle$.

3 Algorithm

XHAIL can compute the solutions of $ILP(B_{E_1}, M, \{E^+, E^-\}_{E_1})$. However how to compute the solutions of $ILP^{DE}(B, M, \langle E_1, E_2\rangle)$ without solving the standard Inductive Logic Programming task constructed from E_1 and E_2 (denoted by $ILP(B_{E_1,E_2}, M, \{E^+, E^-\}_{E_1,E_2})$) ? This section addresses this question. Before that we define the following terms which will be needed for the discussion.

Definition 3
$H_1 \leq H_2$ Two answer set programs H_1 and H_2 are related by "$\leq$" (denoted as $H_1 \leq H_2$) if and only if H_1 can be transformed into H_2 by either adding new rules to H_1 or by adding new literals in the body of the existing rules.

Definition 4
Minimality A solution H of $ILP(B, M, E)$ is **minimal** *iff* $\nexists H' < H$ in $L(M)$ that solves $ILP(B, M, E)$.

Definition 5
Distinctness A series of examples $E_i\langle O_i, E_i^+, E_i^-\rangle, i = 1...n$ are said to be *distinct iff*, $\Delta(B \cup O_1 \cup ... \cup O_n, M, \cup_{i=1}^n E_i^+, \cup_{i=1}^n E_i^-) = \{\cup_{i=1}^n \triangle_i | (\triangle_1, ..., \triangle_n) \in \Delta(B \cup O_1, M, E_1^+, E_1^-) \times ... \times \Delta(B \cup O_n, M, E_n^+, E_n^-)\}$. A series of examples $E_i\langle O_i, E_i^+, E_i^-\rangle, i = 1...n$ are said to be *mutually distinct iff* all subsets of the examples are *distinct*.

Now consider the two examples E_1 and E_2 . Since E_1 and E_2 are *distinct* examples constructed from two different $\langle x, y\rangle$ pairs, by definition, $\Delta(B \cup O_1 \cup O_2, M, \cup_{i=1}^2 E_i^+, \cup_{i=1}^2 E_i^-) = \{\triangle_1 \cup \triangle_2 | (\triangle_1, \triangle_2) \in \Delta(B \cup O_1, M, E_1^+, E_1^-) \times \Delta(B \cup O_2, M, E_2^+, E_2^-)\}$. Thus, for any solution $\langle H_I, H_G, \triangle\rangle$ of $ILP(B \cup O_1 \cup O_2, M, \cup_{i=1}^2 E_i^+, \cup_{i=1}^2 E_i^-)$, $\exists \triangle_1 \in \Delta(B \cup O_1, M, E_1^+ \cup E_1^-)$ and $\exists \triangle_2 \in \Delta(B \cup O_2, M, E_2^+ \cup E_2^-)$ such that,

$$H_G(\triangle) = H_G(\triangle_1) \cup H_G(\triangle_2) \geq H_I$$

This property allows us to search for H_I's without solving $ILP(B \cup O_1 \cup O_2, M, \cup_{i=1}^2 E_i^+, \cup_{i=1}^2 E_i^-)$ directly. The search procedure can be briefly described as follows: For any choice of $(\triangle_1, \triangle_2)$ pair, first find all the minimal $H \leq H_G(\triangle_1) \cup H_G(\triangle_2)$ that solves E_1 and then expand those minimally, with respect to E_2 and E_1 alternatively, until all the minimal H_I's that solves both E_1 and E_2 are found. To find all the H_I one simply needs to iterate over all possible $(\triangle_1, \triangle_2)$ pairs which can be computed from $ILP(B \cup O_1, M, E_1^+, E_1^-)$ and $ILP(B \cup O_2, M, E_2^+, E_2^-)$ individually.

It should be noted that it is possible to have $H_G(\triangle') = H_G(\triangle'')$, even though $\triangle' \neq \triangle''$. Thus, the above search procedure can be optimized by iterating over pairs of generalizations instead of iterating over the abducibles. Another drawback of the above search procedure is that the search results of $(H_G^1(\triangle_1), H_G^2(\triangle_2))$ do not give any information for the search initiated on $(H_G^1(\triangle_1'), H_G^2(\triangle_2'))$. In every iteration it starts from scratch. However, if we remember the solutions of $ILP^{DE}(B, M, E_1)$, we can use those as lower bounds for finding the solutions of $ILP^{DE}(B, M, \langle E_1, E_2\rangle)$. This is because, if H_I is a minimal solution of $ILP^{DE}(B, M, \langle E_1, E_2\rangle)$, then H_I also solves $ILP^{DE}(B, M, E_1)$

and there exists a $\langle H_I^1, H_G^1, \triangle_1 \rangle \in ILP^{DE}(B, M, E_1)$ such that $H_I^1 \leq H_I$. Thus, for the iteration $(H_G^1(\triangle_1), H_G^2(\triangle_2))$, one can search if some $H_I^1 \leq H_G^1(\triangle_1)$ can be expanded by either expanding some rules in H_I^1 or by adding new rules from the remainder of $H_G^1(\triangle_1) \cup H_G^2(\triangle_2)$ or both to solve E_2 along with E_1. Theorem 1 formalizes this idea.

Theorem 1
For any solution $\langle H_I, H_G, \triangle \rangle$ of $ILP^{DE}(B, M, \langle E_1, ..., E_n \rangle)$ there exists a solution $\langle H_I', H_G', \triangle' \rangle$ of $ILP^{DE}(B, M, \langle E_1, ..., E_{n-1} \rangle)$ and a generalization H_G'' in $ILP^{DE}(B, M, E_n)$ such that, $H_I' \leq H_I \leq H_G' \cup H_G''$, when $H \cup B \cup O_i$ is *stratified* for any choice of $i \in \{1, ..., n\}$ and $H \in \{H_G, H_G', H_G''\}$. Here, O_i is the *observation* from E_i. ■

With this in mind, the algorithm for finding the solutions of $ILP^{DE}(B, M, \{E_1, E_2, ..., E_n\})$ is described in Algorithm 1. The proof of the theorem is in Appendix A.

Example

In this subsection we describe how our algorithm computes a solution to the running example $ILP^{DE}(B, M, \langle E_1, E_2, E_3 \rangle)$ from Table 1. Here B contains all the constants of type $op1$, *direction* and *time* and M is the one described in Table 3 .

Initialization: First the *stack* is filled with the output from $XHAIL(B, M, E_1)$. In section 1, we have seen that the output contains only one tuple. The following block shows the content of the stack after initialization.The underlined part denotes H_I, where H_G is the entire program.

$$\underline{holdsAt(relativeposition(X, Y, right), T)}$$
$$\leftarrow holdsAt(relativeposition(Z, X, above), T),$$
$$holdsAt(relativeposition(Y, Z, left), T).$$

Iteration 1: In iteration 1, the hypothesis on the top (denoted as $Top\langle H_I^{Top}, H_G^{Top} \rangle$) of the stack is popped. One can see that the hypothesis H_I^{Top} does not cover E_2. So, the algorithm tries to find an expansion of it which solves E_2 and E_1 both. For that it first finds $H_G(B, M, E_2)$ and creates a new *refinement stack* with lower bound (H_I^{Top}) - upper bound ($H_G^{Top} \cup H_G^{Top}$) pairs as shown below:

$$\underline{holdsAt(relativeposition(X, Y, right), T)}$$
$$\leftarrow holdsAt(relativeposition(Z, X, above), T),$$
$$holdsAt(relativeposition(Y, Z, left), T).$$

It may be noted that $H_G(B, M, E_2)$ is empty as E_2 does not contain any positive example, so the stack contains only and exactly the *Top*. Next it pops the *refinement stack* and tries to find the minimal extensions of the *Top* that covers E_2. There are two such minimal extensions , H', H'' and both of them are pushed to the *refinement stack*.

Algorithm 1: I^2XHAIL

Data: An instance of $ILP^{DE}(B, M, \{E_1, \ldots, E_n\})$
Result: A solution to the problem

```
/* initialize a stack with the solutions of ILP(B,M,E_1)                    */
stack = XHAIL(ILP(B,M,E_1));
while stack is not empty do
    /* pop the hypothesis from the top                                      */
    ⟨H_I, H_G⟩ = stack.pop();
    /* get an example E_i such that B∪H_I∪O_i ⊬ E_i^+ or B∪H_I∪O_i ⊢ E_i^- */
    E_i = nextUncoveredExample(H_I);
    /* No such example exists                                               */
    if E_i is null then
        /* found a solution                                                 */
        return H_I.
    else
        /* Find expansions of H_I that also solves E_i                      */
        refinementsStack = <> ;
        /* support set denotes the set of examples from which <H_i, H_G>
           is created                                                       */
        supports = supportSet(H_I) ∪ {E_i};
        /* compute a set of lower bound-upper bound pairs for the search
           space.                                                           */
        H_G(E_i) = findGeneralizatons(B, M, E_i);
        foreach H in H_G(E_i) do
            push ⟨H_I, H_G ∪ H⟩ to refinementsStack
        while refinementsStack is not empty do
            /* get a candidate lower bound-upper bound pair                 */
            ⟨H'_I, H'_G⟩ = refinementsStack.pop();
            /* get an example from supports that is not covered by H'_I     */
            E_j = nextUncoveredExampleFromS(H'_I, supports);
            if E_j is null then
                /* if no such example exists then we found a solution to
                   the subproblem. Push it to the stack.                    */
                push ⟨H'_I, H'_G⟩ to stack;
            else
                /* Expand H'_I minimaly along H'_G so that it covers E_j    */
                expansions = expandMinimal(⟨H'_I, H'_G⟩, B, E_j);
                /* Push all expansions in the refinementsStack for further
                   updates.                                                 */
                foreach ⟨H''_I, H''_G⟩ in expansions do
                    refinementsStack.push(⟨H''_I, H''_G⟩)
```

$$H' = \left\{ \begin{array}{l} \underline{holdsAt(relativeposition(X,Y,right),T)} \\ \quad \underline{\leftarrow holdsAt(relativeposition(Z,X,above),T),} \\ \quad holdsAt(relativeposition(Y,Z,left),T). \end{array} \right\}$$

$$H'' = \left\{ \begin{array}{l} \underline{holdsAt(relativeposition(X,Y,right),T)} \\ \quad \leftarrow holdsAt(relativeposition(Z,X,above),T), \\ \quad \underline{holdsAt(relativeposition(Y,Z,left),T).} \end{array} \right\}$$

The algorithm then goes on popping the top of the *refinement stack*, say H'. Since H' solves both E_1 and E_2 the condition on line 16 of Algorithm 1 is satisfied and H' is pushed into the main stack. Similarly, H'' is popped next and pushed to the main stack. At this point *refinement stack* becomes empty and iteration 1 exits as it has discovered all the minimal extensions of Top. The stack now contains H'' on top of H'.

Iteration 2: In the next iteration the algorithm pops $\langle H''_I, H''_G \rangle$ which is currently at the top of the stack. The next problem that it does not solve is E_3. It then computes $H_G(B, M, E_3)$ which contain only one element,

$$H''' = \left\{ \begin{array}{l} holdsAt(relativeposition(X,Y,below),T) \\ \quad \leftarrow holdsAt(relativeposition(Z,Y,below),T), \\ \quad holdsAt(relativeposition(X,Z,right),T). \end{array} \right\}$$

It then pushes $\langle H''_I, H''_G \cup H''' \rangle$ to the refinement stack and finds the minimal expansions of H''_I within the bound of $H''_G \cup H'''$. There will be only one such expansion, H^{final} which will then be pushed into the refinement stack and finally into the main stack. Since H^{final} solves all three examples, the algorithms terminates returning H^{final} as the solution.

$$H^{final} = \left\{ \begin{array}{l} holdsAt(relativeposition(X,Y,right),T) \\ \quad \leftarrow holdsAt(relativeposition(Y,Z,left),T). \\ holdsAt(relativeposition(X,Y,below),T) \leftarrow . \end{array} \right\}$$

On the Minimality of the Solution

The solution returned by algorithm 1 may not be minimal. This is because if H_I is expanded minimally to H'_I to solve a new example E, it does not ensure that H'_I is minimal with respect to the relevant subproblem. An example of this is the following: $B = \{\}$, $E_1 = \langle \{p.,b.,c.\}, \{a\}, \{\} \rangle$, $E_2 = \langle \{b.\}, \{\}, \{a\} \rangle$, $E_3 = \langle \{c.\}, \{a\}, \{\} \rangle$, and $M = \{\#modeh\ a, \#modeb\ b, \#modeb\ c, \#modeb\ p\}$. There are two solutions in $ILP^{DE}(B, M, \langle E_1, E_2 \rangle)$: $H_1 = \{a \leftarrow c.\}$ and $H2 = \{a \leftarrow p.\}$. If H_2 is expanded first, it will produce $\{a \leftarrow p., a \leftarrow c.\}$ as the solution of $ILP^{DE}(B, M, \langle E_1, E_2, E_3 \rangle)$ and since it covers all the examples, it will be returned as the solution. However, only $\{a \leftarrow c.\}$ is sufficient to cover E_1, E_2, E_3. Thus the output is not minimal. The minimal solution can be found by computing all the solutions to $ILP^{DE}(B, M, \langle E_1, E_2, E_3 \rangle)$ and then discarding the ones which have a compressed version of it already in $ILP^{DE}(B, M, \langle E_1, E_2, E_3 \rangle)$. However, algorithm 1 prefers efficiency over minimality and returns the first solution found.

4 Related Work

In recent years the field of Inductive logic programming has seen major advancements in many of its areas. Different ILP algorithms have been proposed (Ray 2009; Athakravi et al. 2013; Law et al. 2014; Athakravi et al. 2015; Katzouris et al. 2015; Kazmi et al. 2017; Schüller and Kazmi 2017). Researchers have analyzed various kinds of "good" rules that cannot be learned with the current definition of entailment (called "cautious inference") and proposed an alternative to that, named as "brave inference". ILP Algorithms have thus been proposed that can do only "brave inference" (Otero 2001) or both (Sakama 2005; Sakama and Inoue 2009; Law et al. 2015). Efforts have also been made to learn answer set programs that not only contain Horn clauses but also choice rules and constraints (Law et al. 2015). With these developments and the various systems that have been produced with these researches, people have successfully applied the paradigm of Inductive logic programming to various areas (Gulwani et al. 2015; Mitra and Baral 2016). And with these exposures to different applications, several changes are being made to the paradigm of ILP.

Recently (Law et al. 2016) proposed context dependent learning for *ordered* answer set programs. Due to lack of space we do not discuss learning ordered answer set programs here. Interested readers can refer to (Law et al. 2016). The definition of context dependent learning in this paper is an adaptation of their definition for standard ILP setting. It should be noted that even though the concept of context depending learning was proposed in (Law et al. 2016), to solve the problem their method converts it to a standard ILP problem using choice rules. Here, we have made the first attempt to solve the problem in its original form.

In this work, we deal with the situation where there are many small distinct examples $\{(x_1, y_1), ..., (x_n, y_n)\}$. Another situation where scalability is needed, is when there is a single but large example. Works in (Katzouris et al. 2015; Katzouris et al. 2017) talk about this situation. Our work is also related to the work in logical vision (Dai et al. 2015) that aims to learn symbolic representation of simple geometric concepts.

5 Experiments

We have applied our algorithm on two datasets. They are discussed below:

Task 6: Lists/Sets	**Task 17: Path finding**	**Task 10: Indefinite reasoning**
Sandra picked up the football there.	The office is east of the hallway.	Fred is either in the school or the park.
Sandra journeyed to the office.	The kitchen is north of the office.	Mary went back to the office.
Sandra took the apple there.	The garden is west of the bedroom.	Bill is either in the kitchen or the park.
Sandra discarded the apple.	The office is west of the garden.	Fred moved to the cinema.
What is Sandra carrying?	How do you go from the kitchen to the garden?	Is Bill in the office?

Table 4: Example question answering tasks from bAbI dataset

Question Answering

Recently a group of researchers from Facebook has proposed a question answering challenge (Weston et al. 2015) containing 20 different tasks. Table 1 and 4 shows examples of such tasks.

TASK	Time	Rules	Acc
1: Single Supporting Fact	3	10	100
2: Two Supporting Facts	3	2	100
3: Three Supporting facts	_	_	100
4: Two Argument Relations	2	8	100
5: Three Argument Relations	6	20	100
6: Yes/No Questions	_	_	100
7: Counting	5	14	100
8: Lists/Sets	4	8	100
9: Simple Negation	4	13	100
10: Indefinite Knowledge	9	21	100

TASK	Time	Rules	Acc
11: Basic Coreference	4	5	100
12: Conjunction	_	_	100
13: Compound Coreference	_	_	100
14: Time Reasoning	4	4	100
15: Basic Deduction	4	1	100
16: Basic Induction	4	1	93.6
17: Positional Reasoning	4	26	100
18: Size Reasoning	4	4	100
19: Path Finding	17	2	100
20: Agent's Motivations	2	6	100

Table 5: Performance on the set of 20 tasks. The tasks for which training is not required is marked with '-'. *Running time* is measured in *minutes*.

Each task contains 1000 or more such stories in the training data. The goal is to build a system that uniformly solves all the tasks.

The work of (Mitra and Baral 2016) has shown how Inductive logic programming can be used to solve the tasks. Their method can be summarized as follows: Given the input containing a story and a question, first translate it to an Answer Set Program using a natural language parser and some handwritten rules, then use some knowledge to answer the question. In the training phase, learn the necessary knowledge. They have used *XHAIL* system to learn the knowledge. However, *XHAIL* could not scale to the entire dataset. So they have divided the dataset. For each task their method takes a bunch of examples together, learns from the bunch using *XHAIL*, adds the learned hypothesis back to the background knowledge and then takes the next bunch to learn from. Since knowledge learned from a group of examples is never updated again, they had to manually find a group size that will work for this dataset. The group size depended on the task and clearly it might happen that for some new task there does not exist a group size to which *xhail* can scale. In this work, we reuse the dataset, their mode declarations and have found that our algorithm can learn all the knowledge given the input $ILP^{DE}(B, M, D_{task})$, where D_{task} contains all the 1000 examples of a task. Table 5 shows the time it has taken, the number of rules learned for each task and the accuracy for each task. Our system has achieved the same accuracy as that of (Mitra and Baral 2016).

Semantic Parsing We have done further experiments with the task of semantic parsing. We took all the unique sentences in the training dataset of (Weston et al. 2015) and the corresponding parse tree of the sentences and then trained an ILP system to do the conversion from scratch. Table 6 shows an example of this task. The training dataset contains 5458 such examples. Our system learned a collection of 165 rules in 128 minutes from the training data which accurately parsed all the sentences in the test data.

Handwritten Digit Recognition

The MNIST dataset (LeCun 1998) contains images of handwritten digits. Each image is a 28×28 matrix and is labeled with a number between 0 to 9 denoting the digit it represents. The value of a cell (pixel) in the matrix (image) ranges between 0 (black) to 255 (white) capturing the darkness at that point. In this experiment we use our ILP algorithm to learn rules that identifies digits. For that we represent the images in the following way:

Sentence
Daniel journeyed to the bathroom.
ASP Representation O_i
index(1..5). lemma(1,daniel). pos(1,nn). lemma(2,journey). pos(2,vbd). lemma(3,to). pos(3,to). lemma(4,the). pos(4,dt). lemma(5,bathroom). pos(5,nn).
Positive Examples E_i^+
arg1(journey01,daniel), arg2(journey01,bathroom) .
Positive Examples E_i^-
any possible output that is not in E^+.

Table 6: An example from the semantic parsing task. For each word in the sentence the representation contains its lemma and pos tag, which are obtained using Stanford parser .

1. First, we divide all cell value by 255 so that the value of each cell is in the range of $[0,1]$.
2. For each 4×4 non-overlapping submatrix we create a super-pixel whose value is the sum of the all the pixels in that region. This gives a 7×7 size matrix representation of the original image. Note that in this reduced matrix, each cell value ranges between 0 to 16.
3. If the value of a super-pixel from the 7×7 matrix is less than 2 we consider it to be in the *off* state. If the value is more than or equal to 5 we consider it be in the *on* state. The original image is then described as two disjoint sets: 1) a set of positions where the state of the super-pixel is *off* and 2) another set where all the super-pixel are *on*.

We learn rules on this representation. Each learned rule for a digit d simply says, if the super-pixels in certain positions are *off* and are *on* for some other positions then the image represents the digit d. The training data in the MNIST dataset contains a total of $60,000$ images with approximately $6,000$ images for each digit. To learn the rules for each digit we take all the examples of that digit and take equal amount of images that represent other digits and pass that to our algorithm. Table 7 shows the number of rules learned for each digit and the performance on the test data. Except for the digit 1, it takes 160 hours to learn the rules for each digit.

Digit	#Rules	#Test Examples	Acc(%)	Digit	#Rules	#Test Examples	Acc(%)
0	3,021	980	60.91	5	3,459	891	42.65
1	444	1134	95.85	6	2,621	958	65.03
2	4,606	1032	32.95	7	2,430	1028	63.52
3	3,661	1010	49.80	8	3,237	978	54.50
4	3,416	982	49.59	9	2,382	1009	69.18

Table 7: Performance on handwritten digit recognition tasks. For each digit, column 2 shows the numbers of rules learned, the number instances of that digit in the test set and the percentage of instances correctly classified.

As the Table 7 suggests the performance on handwritten digit recognition is quite poor in comparison to the state-of-the-art neural network classifier (Wan et al. 2013) that achieves 99.79% accuracy on this dataset. The number of rules column in Table provides insights on this high error rates. Consider the example of digit 0. If there are 5000 instances of digit 0 and the algorithm outputs $3,021$ rules that means the representation that we have chosen does not allow good generalization. However, the representation seems to work quite well for the digit 1.

An important lesson learned from this experiment is that even though it takes a small amount

of time to perform a hypothesis refinement when finding a solution H for $\langle E_1, ..., E_i \rangle$ from a solution of $\langle E_1, ..., E_{i-1} \rangle$, the algorithm needs to verify if H explains all of $\{E_1, ..., E_i\}$ before it can proceed to the next iteration. If the size of H is big (such as the case for digit recognition) and too many refinements are taking place then the algorithm spends a lot of time in the verification phase. An important future work will be to optimize this step by identifying which examples could have been affected if a hypothesis goes through refinement. Nevertheless, the algorithm is able to output a solution and does not blow up when a problem of this size is given as input. The dataset associated with all the experiments and the learned rules are available at `https://goo.gl/k6AEEz`. All experiments were performed on an intel i7 machine with 12 GB RAM.

6 Conclusion

Earlier days of Artificial Intelligence have seen many handwritten rule based systems. Later those were replaced by better performing machine learning based systems. With the advancements of knowledge representation and reasoning languages, a natural question arises, "if machines can learn logic programs, can it achieve better accuracy than existing statistical machine learning methods such neural networks?" It should be noted that the system of (Mitra and Baral 2016) achieved better results than the existing deep learning models on the bAbI dataset. To further explore this possibility we need to focus on the task of learning of logic programs and need to develop systems that can learn from large datasets. In this paper, we have made an attempt towards that.

Acknowledgments

We are grateful to Stefano Bragaglia for making the code of XHAIL publicly available which is reused in the development of our system. We would also like to thank the reviewers for their insightful comments. This work has been supported by the NSF grant 1750082.

References

Athakravi, D., Alrajeh, D., Broda, K., Russo, A., and Satoh, K. 2015. Inductive learning using constraint-driven bias. In *Inductive Logic Programming*, pp. 16–32. Springer, Cham.

Athakravi, D., Corapi, D., Broda, K., and Russo, A. 2013. Learning through hypothesis refinement using answer set programming. In *International Conference on Inductive Logic Programming*, pp. 31–46. Springer.

Dai, W.-Z., Muggleton, S. H., and Zhou, Z.-H. 2015. Logical vision: Meta-interpretive learning for simple geometrical concepts. In *ILP (Late Breaking Papers)*, pp. 1–16.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *ICLP/SLP*, Volume 88, pp. 1070–1080.

Gulwani, S., Hernandez-Orallo, J., Kitzelmann, E., Muggleton, S., Schmid, U., and Zorn, B. 2015. Inductive programming meets the real world. *Communications of the ACM 58,* 11, 90–99.

Katzouris, N., Artikis, A., and Paliouras, G. 2015. Incremental learning of event definitions with inductive logic programming. *Machine Learning 100,* 2-3, 555–585.

Katzouris, N., Artikis, A., and Paliouras, G. 2017. Distributed online learning of event definitions. *CoRR abs/1705.02175*.

Kazmi, M., Schüller, P., and Saygin, Y. 2017. Improving scalability of inductive logic programming via pruning and best-effort optimisation. *Expert Systems with Applications*.

Law, M., Russo, A., and Broda, K. 2014. Inductive learning of answer set programs. In *European Workshop on Logics in Artificial Intelligence*, pp. 311–325. Springer, Cham.

Law, M., Russo, A., and Broda, K. 2015. Learning weak constraints in answer set programming. *Theory and Practice of Logic Programming 15,* 4-5, 511–525.

Law, M., Russo, A., and Broda, K. 2016. Iterative learning of answer set programs from context dependent examples. *Theory and Practice of Logic Programming 16,* 5-6, 834–848.

LeCun, Y. 1998. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

Mitra, A. and Baral, C. 2016. Addressing a question answering challenge by combining statistical methods with inductive rule learning and reasoning. In *AAAI*, pp. 2779–2785.

Muggleton, S. 1991. Inductive logic programming. *New generation computing 8,* 4, 295–318.

Muggleton, S. 1995. Inverse entailment and progol. *New generation computing 13,* 3-4, 245–286.

Otero, R. 2001. Induction of stable models. *Inductive Logic Programming*, 193–205.

Ray, O. 2009. Nonmonotonic abductive inductive learning. *Journal of Applied Logic 7,* 3, 329–340.

Sakama, C. 2005. Induction from answer sets in nonmonotonic logic programs. *ACM Trans. Comput. Logic 6,* 2 (April), 203–231.

Sakama, C. and Inoue, K. 2009. Brave induction: a logical framework for learning from incomplete information. *Machine Learning 76,* 1 (Jul), 3–35.

Schüller, P. and Kazmi, M. 2017. Best-effort inductive logic programming via fine-grained cost-based hypothesis generation. *arXiv preprint arXiv:1707.02729*.

Wan, L., Zeiler, M., Zhang, S., Le Cun, Y., and Fergus, R. 2013. Regularization of neural networks using dropconnect. In *International Conference on Machine Learning*, pp. 1058–1066.

Weston, J., Bordes, A., Chopra, S., and Mikolov, T. 2015. Towards ai-complete question answering: a set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*.