Difflog: Learning Datalog Programs by Continuous Optimization

Mukund Raghothaman and Xujie Si and Kihong Heo and Mayur Naik
{rmukund,xsi,kheo,mhnaik}@cis.upenn.edu
University of Pennsylvania

Abstract
An important problem in automated reasoning involves learning logical patterns from structured data. Existing approaches to this task of inductive logic programming either involve solving computationally difficult combinatorial problems or performing parameter estimation in complex statistical relational models. In this paper, we present Difflog, a simple extension of the popular logic programming language Datalog to the continuous domain. By attaching real-valued weights to individual rules, we naturally extend the traditional Boolean semantics of Datalog to additionally associate numerical values with individual conclusions. Rule learning may then be cast as the problem of determining the values of the weights which cause the best agreement between training labels and induced values of output tuples. We propose a novel algorithmic framework to efficiently evaluate Difflog programs with provenance information, which in turn makes it feasible to employ standard numerical optimization techniques such as gradient descent and Newton’s method to the synthesis of logic programs. On a suite of 10 benchmark problems from different domains, Difflog can learn complex programs with recursive rules and relations of arbitrary arity, even with small amounts of noise in the training data.

1 Introduction
Logical (discrete) and statistical (continuous) modes of reasoning have complementary benefits: the logical part promises interpretability, extensibility, and correctness guarantees, while the statistical part offers better robustness in the presence of noise and uncertainty. Many models have been proposed to leverage the benefits of both modes without suffering their drawbacks: they are studied in the field of statistical relational learning, and include stochastic logic programs (Muggleton and others 1996), robust logic (Valiant 1999), probabilistic relational models (Koller 1999), Bayesian logic (Milch et al. 2005), Markov logic networks (Richardson and Domingos 2006), probabilistic soft logic (Bach et al. 2017), and probabilistic Prolog (Raedt, Kimmig, and Toivonen 2007).

The two central problems of interest in the study of these models are inference and learning. While remarkable strides have been made in the area of inference, however, there is a dearth of techniques in the area of learning. Prominent learning efforts include inductive logic programming (ILP) (Muggleton and De Raedt 1994) and program synthesis (Gulwani, Polozov, and Singh 2017). However, in these efforts, the influence of each logical rule considered during learning is discrete: it is either present or absent, which forgoes the benefits of continuous reasoning. In contrast, considerable advances have been made in machine learning by virtue of employing continuous reasoning. Recent ILP systems such as βILP (Evans and Grefenstette 2018) and NEURALLP (Yang, Yang, and Cohen 2017) have demonstrated the promise of leveraging this style of reasoning, but they are fundamentally limited to learning rules of a constrained form, such as fixed arity relations and no recursion.

In this paper, we propose a novel approach and system called Difflog that employs continuous reasoning to learn rich logical rules from data. Difflog can learn complex programs with recursive rules and relations of arbitrary arity even with small amounts of noise in the training data. We target Datalog, a declarative logic programming language that has witnessed applications in a variety of domains including bioinformatics (King 2004; Santos et al. 2012), big-data analytics (Seo, Guo, and Lam 2013; Shkapsky et al. 2016; Halperin et al. 2014), natural language processing (Mooney 1996), networking (Loo et al. 2009), program analysis (Grebenshchikov et al. 2012; Bravenboer and Smaragdakis 2009), and robotics (Poole 1995). Datalog is an appealing target because it is expressive enough—it captures all PTIME problems (Dantsin et al. 2001)—yet concise enough for learning to be practical.

Difflog extends the classical semantics of Datalog to the continuous domain by attaching numerical weights to individual rules. This allows us to apply numeric optimization techniques such as gradient descent and Newton’s method to synthesize Datalog programs, which we formalize as the combinatorial problem of selecting rules from a soup of candidates. Analogous to the training process for deep neural networks, wherein the gradient must be propagated through the network to determine its weights, the training process in Difflog must also propagate the gradient through the rules to determine their weights. However, applying the backpropagation procedure popularly used for this purpose in deep neural networks requires the entire derivation graph of each derived tuple. In contrast, we propose an efficient forward
int **a1, **a2;
int *b1, *b2, *d, *f;
int c1, c2;

b1 = &c1;
b2 = &c2;
a1 = (...) ? &b1 : &b2;
d = *a1;
a1 = &f;
*a1 = d;
a2 = a1;

(a) Example C program. (b) Its points-to graph.

<table>
<thead>
<tr>
<th>Input tuples (EDB)</th>
<th>Output tuples (IDB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr(b1, c1)</td>
<td>pt(a1, b1)</td>
</tr>
<tr>
<td>load(d, a1)</td>
<td>pt(a1, b2)</td>
</tr>
<tr>
<td>store(a1, d)</td>
<td>pt(a1, f)</td>
</tr>
<tr>
<td>copy(a2, a1)</td>
<td>pt(a2, b1)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(c) Input (program) and output (points-to graph) tuples.

\[
\begin{align*}
R_1 : & \quad pt(p, q) \leftarrow addr(p, q). \\
R_2 : & \quad pt(p, r) \leftarrow copy(p, q), pt(q, r). \\
R_3 : & \quad pt(p, s) \leftarrow load(p, q), pt(q, r), pt(r, s). \\
R_4 : & \quad pt(r, s) \leftarrow store(p, q), pt(p, r), pt(q, s).
\end{align*}
\]

(d) Points-to analysis derived from input/output tuples.

Figure 1: An example C program, its points-to graph, tuple-based representations of the program and the graph, and pointer analysis synthesized by DIFFLOG that computes the graph given the program.

In fact, this problem is undecidable—for instance, the “(...)” can be arbitrary code—and any pointer analysis necessarily over-approximates, i.e., it derives spurious points-to facts; for instance, Andersen’s analysis incorrectly concludes that \( b_2 \) points to \( c_1 \). This information is represented as a points-to graph shown in Figure 1(b) where true (resp. false) points-to facts are denoted by solid (resp. dashed) edges.

Our goal is to learn the rules of Andersen’s analysis expressed by the Datalog program in Figure 1(d) given the input/output data in Figure 1(c). The input data (also called extensional database or EDB) represents relevant facts about the input C program, such as tuples \( addr(x, y) \) for statements of the form \( x = &y \), tuples \( load(x, y) \) for statements of the form \( x = *y \), and so on. The output data (also called intentional database or IDB) represents exact points-to facts, namely, tuples \( pt(x, y) \) denoting that \( x \) may point to \( y \). These tuples correspond to the solid edges in Figure 1(b).

There are several challenges in learning the above program. First, it includes self-recursive and mutually-recursive rules. For instance, rule \( R_2 \) states that if the program contains a statement \( p = q \) and if we have deduced that \( q \) points to \( r \) then we may deduce that \( p \) points to \( r \). Second, the rules follow patterns with subtle variations, making it challenging to determine the space of candidate rules to consider. For instance, rules \( R_3 \), \( R_2 \), and \( R_3 \) follow the ubiquitous chain pattern whose general form is \( r_0(x, y) := r_1(x, t_1), r_2(t_1, t_2), ..., r_n(t_n-1, y) \). But rule \( R_4 \) does not obey this pattern.

Many existing approaches do not support recursion, only support binary relations, and only target rules that have a constrained form, such as the chain pattern. In contrast, DIFFLOG supports recursive rules and relations of arbitrary arity. Moreover, it generates a rich soup of candidate rules through a procedure called k-augmentation (see Sec. 3.3). It starts out with the chain pattern and applies up to \( k \) edits to generate increasingly rich variants; when \( k = \infty \), the soup contains all possible Datalog rules, although a small \( k \) suffices in practice. For instance, the pattern of rule \( R_4 \) is generated with \( k = 3 \); the three edits correspond to the three differences of \( R_4 \) compared to \( R_3 \).

Yet another challenge highlighted in the example is that, due to the undecidability of pointer analysis, no Datalog program can capture exact points-to facts for every C program. In this case, we should still learn rules that best approximate the training data. For instance, even though tuples \( pt(b_1, c_2) \) and \( pt(b_2, c_1) \) are excluded from the labeled output, we should learn the Andersen’s analysis rules despite the fact that they end up deriving those tuples. This is possible only by leveraging continuous reasoning. Existing approaches based on discrete reasoning fail to generate any rules in such cases. Finally, the training data may contain noise in the form of mislabeled tuples; similarly, in such cases, we should learn rules that best explain the training data.

DIFFLOG satisfies all of the above criteria, and generates the depicted Datalog program in 500 iterations of gradient descent in a total of 1.5 minutes. In contrast, NEURALLP, which does not support recursion nor the non-chain pattern rule \( R_4 \), learns an approximate program that has 42.8% RMS error even on the training data. Finally, METAGOL (Cropper,
There is a large body of previous research in the field of \textit{inductive logic programming} (ILP) (Muggleton 1991), ranging from foundational theoretical concepts such as \textit{\theta-subsumption} (Plotkin 1970), relative generalization (Plotkin 1972), and refinement (van der Laag and Nienhuys-Cheng 1998), to practical implementations such as FOIL (Quinlan 1990; Quinlan and Cameron-Jones 1993), QuickFOIL (Zeng, Patel, and Page 2014) and Metagol (Cropper and Muggleton 2015). There have also been attempts to synthesize recursive queries by constraint solving (Albarghouthi et al. 2017; Si et al. 2018) and other ideas from program synthesis (Osera and Zdancewic 2015; Polikarpova, Kuraj, and Solar-Lezama 2016; Frankle et al. 2016; Albarghouthi, Guliwani, and Kincaid 2013; Kneuss et al. 2013; Kitzelmann and Schmid 2006). In this context, our paper is an attempt to use numerical optimization instead of combinatorial techniques, with the goal of improved scalability and resilience to noise.

We derived inspiration from previous work on \textit{statistical relational learning}, including frameworks such as Markov Logic Networks (Richardson and Domingos 2006) and ProbLog (Raedt, Kimmig, and Toivonen 2007; Manhaeve et al. 2018). Traditional approaches to learning these models consider structure learning (for example (Kok and Domingos 2005)) separately from weight learning (for example (Lodd and Domingos 2007)). Both these procedures usually involve repeatedly performing inference with different parameter valuations, which is unfortunately \#P-complete in most probabilistic models. Simplifying the complexity of inference—the data complexity of Datalog evaluation, and hence of DIFFLOG as well, is merely \textit{PTIME}-complete—is an important technical component of this paper.

A closely related problem is that of \textit{knowledge-base reasoning} (Nguyen 2017) and systems which use differentiable representations of rules to prove queries. The Neural Theorem Prover (NTP) (Rocktäschel and Riedel 2017; Minervini et al. 2018) is a prominent recent example which, in addition to the rules, also associates “subsymbolic” vector representations with the constants of the program. By only attaching weights to the rules, and leaving the constants to be purely symbolic, DIFFLOG allows for easy transfer between completely disjoint training and testing knowledge bases. Other examples include Neural Prolog (Ding et al. 1996), Logic Tensor Networks (Serafini and d’Avila Garcez 2016), \textsc{NeuralLP} (Cohen 2016; Cohen, Yang, and Mazaitis 2017; Yang, Yang, and Cohen 2017), and \textsc{DLILP} (Evans and Grefenstette 2018). These systems typically handle recursion by unrolling loops to a fixed depth $d_0$. In contrast, as a result of the least-fixed point induced by Equation $2$, DIFFLOG semantics are automatically defined for all recursive programs of arbitrary arity, and furthermore, Equation $3$ automatically captures optimizations such as (Minervini et al. 2018). Finally, the underlying techniques for supporting efficient computation of gradients are fundamentally different: while previous techniques leverage back-propagation in deep neural networks, we instead employ forward propagation based on provenance information.

### 3 Our Framework

#### 3.1 Problem description

We begin with a brief overview of Datalog, as presented in (Abiteboul, Hull, and Vianu 1994). We assume a collection of relations, $\{P, Q, \ldots \}$. Each relation $P$ has an arity $k$, and is a set of tuples $P(v_1, v_2, \ldots, v_k)$, where $v_1, v_2, \ldots, v_k$ are constants. Examples include the relations $\text{pt}(p, q)$, $\text{addr}(p, q)$, etc., and the constants $a1, a2$, etc. from Figure 1. Some relations (EDB) are explicitly provided as part of the input, while the remaining relations (IDB) are implicitly specified by a collection of rules, each of the form:

$$R_h(u_h) : - P_1(u_1), P_2(u_2), \ldots, P_k(u_k),$$

where $P_h$ is an output relation, and $u_h, u_1, u_2, \ldots, u_k$ are vectors of variables of appropriate length. Each rule is a universally quantified logical formula, and is read from right-to-left, with the “$::$” operator treated as implication. For example, rule $R_3$ from Figure 1 may be read as, “If there is a statement in the program of the form $p = \star q$ (load($p, q)$), and $q$ may point to $r$ ($\text{pt}(q, r)$), and $r$ may point to $s$ ($\text{pt}(r, s)$), then $p$ may itself point to $s$ ($\text{pt}(p, s)$)”.

Instantiating a rule’s variables yields a grounded constraint $g$ of the form $P_1(v_1) \land \cdots \land P_k(v_k) \implies P_h(v_h)$. In other words, given the set of antecedent tuples, $A_g = \{P_1(v_1), P_2(v_2), \ldots, P_k(v_k)\}$, the rule produces the conclusion $c_g = P_h(v_h)$. To determine the value of the output relations, we repeatedly apply rules to the known facts and accumulate additional conclusions until nothing further can be derived. Each output tuple is therefore witnessed by at least one derivation tree leading back to the input tuples—at fixpoint, all of these trees may be compactly represented by a derivation graph such as that shown in Figure 2.

Given a Datalog program, and a valuation of its input relations $I$, the query evaluation problem asks to determine the valuation of its output relations $O$. In this paper, we are interested in the query synthesis problem: \textit{Given the input...}
tuples $I$, output tuples $O$, and a set of candidate rules $\mathcal{R}$, can we find a subset of rules, $D \subseteq \mathcal{R}$, such that the output of $D$ on $I$ is equal to $O$?

### 3.2 DiffLog: Extending Datalog with continuous semantics

As a first step to solving the query synthesis problem, we generalize the idea of rule selection. Instead of a binary decision, we associate each rule $R$ with a numerical weight $w_R \in [0, 1]$. One possible way to visualize these weights is as the extent to which they are present in the learned program $D$. Naturally, associating weights with individual rules induces numerical values $v_t$ for each of their conclusions $t$: the key design decision is in fixing how the rule weights $w_R$ determine tuple values $v_t$.

Traditionally, a tuple is produced by a Datalog program if there exists some grounded constraint, all of whose antecedents are true, and of which it is the conclusion. Stated differently, the truth value $b_t$ of the tuple $t$ is the disjunction over all possible rule instantiations $g$ such that $c_g = t$, of the conjunction of its antecedents $A_g = \{t_1, t_2, \ldots, t_k\}$:

$$b_t = \bigvee_g (b_{t_1} \land b_{t_2} \land \cdots \land b_{t_k}). \quad (1)$$

The central idea behind DiffLog is to switch the boolean operations $\lor$ and $\land$ in the above equation with the arithmetic operations $\max$ and $\times$. Combined with the idea to associate rules with weights, we define the value $v_t$ associated with a tuple $t$ as follows:

$$v_t = \max_{\mathcal{D}} (w_{g_1} \times w_{g_2} \times \cdots \times w_{g_n}), \quad (2)$$

where $\mathcal{D}$ ranges over all derivation trees with conclusion $t$, and $g_1, g_2, \ldots, g_n$ are the grounded constraints appearing in this tree, and where $w_{g_j}$ is the weight of the associated rule for each grounded constraint $g$.

For example, tuple $t_1 = \text{pt}(b1, c1)$ in Figure 2 is produced by one application of rule $R_1$, and tuple $t_2 = \text{pt}(d, c1)$ is produced by two applications of $R_1$ and one application of $R_3$. Thus, if their weights are initialized to $w_{R_1} = 0.9$ and $w_{R_3} = 0.8$ respectively, then the corresponding tuples values are $v_{t_1} = 0.9$ and $v_{t_2} = 0.9 \times 0.9 \times 0.8 = 0.648$.

Replacing the operations $\lor$ and $\land$ with $\max$ and $\times$ corresponds to interpreting the Datalog program over the Viterbi semiring instead of the traditional Boolean semiring. As a consequence of this, and because all the rule weights are bounded ($0 \leq w_R \leq 1$) it follows that Equation 2 is well-defined, even in pathological situations where a tuple may have infinitely many derivation trees (Green, Karvounarakis, and Tannen 2007). Furthermore, when appropriately instrumented, classical algorithms to solve Datalog programs, such as the semi-naive evaluator, also work for DiffLog. Finally, we can show that the output values $v_t$ are continuous functions of the rule weights $w_R$, and that the provenance $p_t$ of a tuple provides an efficient mechanism to compute the gradient of $v_t$ with respect to the rule weights, as described in Sec. 3.3.

On the other hand, note that the semantics of DiffLog does not form a probability space: while we compute values $v_t$ for each tuple $t$, we do not generalize them to combinations of tuples. In particular, we have no analogue for quantities such as $\Pr(t_1 \land \neg t_2)$. This choice is deliberate: while the data complexity of determining $v_t$ for a fixed DiffLog program can be shown to be polynomial in the size of the input tuples $I$, the complexity of determining $\Pr(t)$ is $\#P$-complete for even the simplest classes of queries over probabilistic databases (Dalvi and Suciu 2004).

### 3.3 Learning DiffLog programs by numerical optimization

We evaluate DiffLog programs using a modified version of the semi-naive algorithm for Datalog (Abiteboul, Hull, and Vianu 1994). At a high level, at each time step $x \in \{0, 1, 2, \ldots\}$, the evaluator maintains an association between output tuples $t$ and their current candidate values $v_t$. The algorithm repeatedly considers instantiations $g$, all of whose antecedents $A_g = \{t_1, t_2, \ldots, t_n\}$ satisfy $v_{t_i} \geq 0$, and updates the candidate value for $t$:

$$v_t^{x+1} = \max(v_t^x, w_g \times v_{t_1}^x \times v_{t_2}^x \times \cdots \times v_{t_n}^x). \quad (3)$$

To be able to compute the gradients $\nabla v_t$ with respect to the rule weights $w_R$, we also maintain a version of the provenance polynomial $p_t$ for each tuple (Green, Karvounarakis, and Tannen 2007). Informally, the provenance describes how the program concluded that $t$ is an output tuple. We label each input tuple with the polynomial $p_0 = 1$, indicating that their value is independent of any rule weight. Subsequently, after each application of Equation 3, we update $p_t^{x+1}$ as follows:

$$p_t^{x+1} = \begin{cases} p_t^x, & \text{if } v_t^{x+1} = v_t^x, \\ w_g \times p_{t_1}^x \times p_{t_2}^x \times \cdots \times p_{t_n}^x, & \text{otherwise,} \end{cases} \quad (4)$$

where $g$ is the same grounded constraint referred to in the value update expression. Observe that, due to the semantics of the $\max$ function, it suffices to track the lineage of tuples along the winning branch, and hence the provenance polynomial $p_t$ reduces to a compact product of rule weights.

The learning problem for DiffLog can then be seen as determining the value of rule weights $w_R$ which causes the greatest agreement between the expected tuple values $l_t = \mathbb{1}(t \in O)$, and the values $v_t$ produced by the DiffLog program. We cast this as an optimization problem for the L2 loss, $f(w) = \sum_t (v_t - l_t)^2$, and optimize for the optimal values using Newton’s method. To avoid pathological behavior associated with multiplication by zero, we further constrain rule weights $w_R \in [0.01, 0.99]$. We stop the optimization process once the L2 loss drops below 0.01, or once the optimizer has performed 500 iterations, or when the magnitude of the gradient is zero.

Our ultimate goal is to learn discrete logic programs through continuous optimization. As a final step, we therefore reinterpret the produced DiffLog program as a Datalog program by only retaining those rules $R$ which (a) have weight $w_R \geq 0$, for some cutoff value $w_0$, and (b) which are useful, i.e. which contribute to the provenance of some output tuple. The cutoff value $w_0$ is chosen so as to minimize L2 error on the training dataset. It is a straightforward observation that if all rule weights $w_R \geq 0$, then $v_t > 0$ iff $t$ is emitted as an
output tuple. The second condition further reduces the number of rules in the learned program. As we shall demonstrate in Sec. 4.1, this is important in improving generalization.

### 3.4 Implementation details

We will now describe the optimizations needed for the DIFFLOG evaluation algorithm to scale to large datasets. Additional implementation details, such as the choice of the soup of candidate rules \( R \), may be found in Appendix A.

The traditional implementation algorithm for Datalog is the “seminaive” evaluator. Because of the closeness of the semantics of the two formalisms, this algorithm can also be readily translated into our setting as well. The main remaining challenge in DIFFLOG evaluation is then the large number of candidate rules that need to be evaluated in order to effectively choose the correct subset. We observed that, when working with a large number of rules, several of them share portions of their bodies. For example, consider the pair of rules \( R_0(t) := R_1(t_1) \), \( R_2(t_2) \), \( R_3(t_3) \), and \( R_4(t) := R_1(t_1) \), \( R_2(t_2) \), \( R_4(t_4) \). We implemented an efficient evaluation algorithm which structures the candidate rules according to a trie and shares the intermediate results of evaluation between them. We demonstrate the greatly improved scalability of our trie-based evaluator in Fig. 4.

The second performance optimization we have implemented in DIFFLOG is a restricted form of eager projection commonly employed in relational databases. Given the set of input tuples \( I \), the evaluator repeatedly instantiates rules \( R \) to produce grounded constraints \( g \). Starting with a single empty instantiation \( V \) of the variables with weight 1, the evaluator iterates over the literals of the rule, and unifies each variable valuation with each tuple in the present relation, to produce a set of extended variable valuations. Consider the rule \( P(x, w) := P(x, y), P(y, z), P(z, w) \), encountered, for example, while learning the transitive closure of a graph. After processing the first two literals, \( P(x, y) \) and \( P(y, z) \), the set of valuations will have associations for the variables \( x, y \), and \( z \). Notice however, that \( y \) does not appear in any subsequent literal of the rule. We therefore drop \( y \) from each valuation currently under consideration. Each new valuation thus obtained may be associated with multiple previous valuations: the weight of the new valuation is therefore set to the maximum of all obsolete contributing valuations. This valuation collapse significantly improves the performance of the DIFFLOG evaluator.

### 4 Experimental Evaluation

The goals of our experiments are: (a) to determine the accuracy of the DIFFLOG learning algorithm, and compare it to previously published tools in the literature, (b) to estimate how sensitive DIFFLOG is to noise in the training data, and whether it can still learn the correct program in the presence of varying amounts of noise (Appendix B.1), and (c) to measure the scalability of the training process, as a function of the number of candidate rules used for training. To this end, we test DIFFLOG on a suite of 10 benchmarks, 5 of which are from the domain of knowledge discovery, and the remaining from automatic program analysis. We list the essential characteristics of these benchmarks in Table 1. We also ran DIFFLOG on the Countries benchmark of (Bouchard, Singh, and Trouillon 2015) in order to compare our tool with Neural Theorem Provers.

#### 4.1 Accuracy of learned programs

We present the test error achieved by DIFFLOG on our benchmarks in Table 2. We also compare it to the baseline algorithms, NEURALLP (Yang, Yang, and Cohen 2017) and METAGOL (Cropper, Tamaddoni-Nezhad, and Muggleton 2016). All algorithms were run with a timeout of 6 hours on a server with 128 GB of memory and 3 GHz AMD Opteron 6220 processors running Linux 3.2.0. Notice that DIFFLOG is able to learn the correct program for all but two of our benchmarks. In contrast, because of the constrained form of the rules mandated by NEURALLP—relations of arity two, and only non-recursive chain rules—it is not applicable to many of our benchmarks. On the other hand, the combinatorial algorithm employed by METAGOL frequently times out.

The results in Table 2 were obtained after reinterpreting the rules of the learned DIFFLOG program as a traditional Datalog program, as discussed in Sec. 3.3. In Fig. 3, we explore this process in more detail. The output of the DIFFLOG learning algorithm may be viewed as a ranked list of rules. We plot the training and test accuracy achieved by each prefix of this ranked list. Observe that optimum training and test accuracy is simultaneously achieved by approximately the same prefixes of the ranked list. In the reinterpretation process outlined in Sec. 3.3, we choose the smallest set of rules that minimizes training error: by only keeping rules that actually produce output tuples in the training dataset, this acts as a sort of regularization and prevents overfitting.

Finally, we also compare DIFFLOG against Neural Theorem Provers on the Countries dataset (Bouchard, Singh, and Trouillon 2015). The task involves predicting missing tuples in a relational database describing world geography. While knowledge base completion is not the primary purpose of the DIFFLOG system, observe that DIFFLOG outperforms NTP on
### Table 1: Benchmark characteristics. The first five benchmarks are from the domain of knowledge discovery while the remaining five are from program analysis.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Relations</th>
<th># Rules</th>
<th># Training tuples</th>
<th># Test tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Input</td>
<td>Output</td>
<td>Expected</td>
<td>Candidates</td>
</tr>
<tr>
<td>Path</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>Ancestor</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>38</td>
</tr>
<tr>
<td>Animals</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>76</td>
</tr>
<tr>
<td>Samegen</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>154</td>
</tr>
<tr>
<td>Knights Move</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>Andersen</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>Escape</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>26</td>
</tr>
<tr>
<td>Modref</td>
<td>7</td>
<td>5</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>1-Call Site</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>94</td>
</tr>
<tr>
<td>Polysite</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>289</td>
</tr>
</tbody>
</table>

### Table 2: F1-score achieved by DIFFLOG and the baseline learning algorithms on our benchmarks. N/A denotes “not applicable”. Timeout is 6 hours.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>F1-Score on Test Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path</td>
<td>1.00 N/A timeout</td>
</tr>
<tr>
<td>Ancestor</td>
<td>1.00 0.88 timeout</td>
</tr>
<tr>
<td>Animals</td>
<td>1.00 0.65 timeout</td>
</tr>
<tr>
<td>Samegen</td>
<td>1.00 0.00 timeout</td>
</tr>
<tr>
<td>Knight Move</td>
<td>1.00 1.00 1.00</td>
</tr>
<tr>
<td>Andersen</td>
<td>1.00 N/A timeout</td>
</tr>
<tr>
<td>Escape</td>
<td>1.00 0.84 timeout</td>
</tr>
<tr>
<td>Modref</td>
<td>1.00 N/A timeout</td>
</tr>
<tr>
<td>1-Call Site</td>
<td>0.97 N/A timeout</td>
</tr>
<tr>
<td>Polysite</td>
<td>1.00 N/A 0.93</td>
</tr>
</tbody>
</table>

### Table 3: AUC-PR on Countries and comparison with Neural Theorem Provers. We report the average of three experiments.

<table>
<thead>
<tr>
<th>Countries</th>
<th>AUC-PR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DIFFLOG</td>
</tr>
<tr>
<td>(S_1)</td>
<td>97.7</td>
</tr>
<tr>
<td>(S_2)</td>
<td>94.9</td>
</tr>
<tr>
<td>(S_3)</td>
<td>89.7</td>
</tr>
</tbody>
</table>

4.2 Scalability of training process

To study the scalability of the DIFFLOG learning process, we measured two quantities: first, the time taken to solve the DIFFLOG program and compute gradients, i.e. the time per iteration of numerical optimization (Fig. 4), and second, the number of iterations needed to converge to the final solution (Fig. 5 of Appendix B). Observe that the trie-based evaluator significantly outperforms the traditional “seminaive” evaluation strategy while the number of iterations required for convergence remains roughly constant, suggesting that

### 5 Conclusion

We presented an approach and system called DIFFLOG to learn Datalog programs from input-output data. Inspired by the success of continuous reasoning in machine learning, DIFFLOG extends Datalog semantics with numerical weights on individual rules, which enables us to apply numeric optimization techniques to synthesize Datalog programs. Our approach leverages query provenance information to efficiently forward-propagate the gradient and learn weights. We also describe a fast evaluation algorithm, based on tries, which outperforms the traditional seminaive evaluation algorithm of Datalog. We demonstrated that DIFFLOG is capable of learning complex Datalog programs with recursive rules and relations of arbitrary arity, even with small amounts of noise in the training data. It thereby targets a richer class of logic programs than state-of-the-art systems, including those based on discrete reasoning as well as those based on continuous reasoning.

In future work, we plan to extend DIFFLOG to address useful Datalog extensions such as aggregation and stratified nega-
tion. Our formulation of the problem as selecting rules from a soup of candidates facilitates supporting such extensions. Another important direction concerns handling black-box predicates, including so-called invented predicates that are constructed using Datalog rules themselves, as well as foreign functions that are constructed outside the Datalog evaluation sub-system.

References
Proceedings of the 11th European Conference on Principles and Practice of Knowledge Discovery in Databases, PKDD, 200–211. Springer.


A Framework

A.1 Choice of the Viterbi semiring

The well-definedness of Equations 2 and 3, even in the presence of arbitrary recursion patterns, is a consequence of the \( \omega \)-continuity of the Viterbi semiring, \(([0, 1], \max, \times, 0, 1)\). Furthermore, semantics of DIFFLOG and classical Datalog is closely related by the following theorem:

**Theorem 1.** Let \( \mathcal{R} \) be a set of DIFFLOG rules, and \( \mathbf{w} \) be an assignment of weights to each of the rules. Let \( P \) be the classical Datalog program consisting of only those rules \( R \in \mathcal{R} \) such that \( w_R > 0 \). Then, for each input database \( D \) and for each potential output tuple \( t \), the value \( v_t \) of the DIFFLOG program satisfies \( v_t \geq 0 \) iff \( t \) is in the output of the classical program \( P(D) \).

This also precludes other possible choices for the semiring. Furthermore, we can also show that the output values \( v_t \) vary continuously with the value of the weight vector \( \mathbf{w} \).

A.2 Choice of candidate rules \( \mathcal{R} \)

The effectiveness of the DIFFLOG search depends on the expressiveness of the set of candidate rules \( \mathcal{R} \). In our experiments, we obtain this set by a process of augmentation, which we now describe. Our motivation is that the rules of Datalog programs tend to be structurally similar to each other, and that small syntactic modifications of one plausible candidate rule can yield another. We therefore start with a set of seed rules, and repeatedly replace relations, variables, and insert additional variables into the bodies of the candidate rules to produce new candidate rules. We keep all candidate rules which are thus obtained, and which are at an edit distance of at most 5 from the following “chain rules”:

\[
P_1(x, y) \leftarrow P_2(x, y), \\
P_1(x, z) \leftarrow P_2(x, y), P_3(y, z), \text{ and} \\
P_1(x, w) \leftarrow P_2(x, y), P_3(y, z), P_4(z, w),
\]

where \( P_1, P_2, P_3, \) and \( P_4 \) are arbitrarily initialized relations.

In particular, we used this collection of rules while obtaining the results in Table 2, but we also go on to explore the effect of drastically increasing the space of available rules, and its effect on evaluation time and rate of convergence in the experiments of Sec. 4.2 and Appendix B respectively.

B Experimental Evaluation

B.1 Sensitivity to training noise

Next, we measured the ability of DIFFLOG to learn programs in the presence of noise. We flipped the truth values of a randomly selected subset of the output tuples in the training data, and measured the final training error and resulting test error of the learned DIFFLOG program. We present these results in Table 4. Observe that, despite the training noise, and hence the training error being non-zero, the optimizer is still able to learn the correct program and generalize to the test data.

<table>
<thead>
<tr>
<th>Proportion of Noise</th>
<th>F1-Score Train</th>
<th>F1-Score Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>0.05</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>0.10</td>
<td>0.95</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 4: F1-score of training and test set achieved by DIFFLOG on Path with noisy training data.