Difflog: Learning Datalog Programs by Continuous Optimization

Mukund Raghothaman
rmukund@cis.upenn.edu

Richard Zhang
rmzhang@cis.upenn.edu

Xujie Si
xsi@cis.upenn.edu

Kihong Heo
kheo@cis.upenn.edu

Mayur Naik
mhnaik@cis.upenn.edu

Abstract

Statistical relational models that combine logical and statistical reasoning offer a variety of benefits. A central problem in using such models concerns learning rich logical rules from data. Existing approaches are hindered by employing discrete reasoning during learning. We propose DIFFLOG, an approach and system based on continuous reasoning. It extends Datalog, a popular logic programming language, 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. Our approach is inspired by the success of continuous reasoning in machine learning, but differs fundamentally by leveraging provenance information that is naturally obtained during Datalog evaluation. On a suite of 10 benchmarks 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 [30], robust logic [45], probabilistic relational models [22], Bayesian logic [24], Markov logic networks [40], probabilistic soft logic [5], and probabilistic Prolog [39].

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) [28] and program synthesis [17]. However, in these efforts, the influence of each logical rule considered during learning is discrete: it is either present or absent, which undermines the benefits of continuous reasoning. In contrast, considerable advances have been made in machine learning by virtue of using continuous reasoning. Recent ILP systems such as δILP [13] and NEURALLP [47] 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.

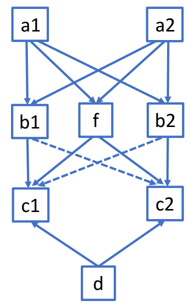
In this paper, we propose a novel approach and system called DIFFLOG that employs continuous reasoning to learn rich logical rules from data. DIFFLOG can learn complex programs with recursive rules and relations of arbitrary arity even with small amounts of noise in the training data. We target Datalog, a declarative logic programming language that has witnessed applications in a variety of domains including bioinformatics [19, 41], big-data analytics [18, 42, 43], natural language

Preprint. Work in progress.

```
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.

Input tuples (EDB)	Output tuples (IDB)
addr(b1, c1)	pt(a1, b1)
load(d , a1)	pt(a1, b2)
store(a1, d)	pt(a1, f)
copy(a2, a1)	pt(a2, b1)
...	...

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

R_1 : $\mathsf{pt}(p, q)$:− $\mathsf{addr}(p, q)$.
R_2 : $\mathsf{pt}(p, r)$:− $\mathsf{copy}(p, q), \mathsf{pt}(q, r)$.
R_3 : $\mathsf{pt}(p, s)$:− $\mathsf{load}(p, q), \mathsf{pt}(q, r), \mathsf{pt}(r, s)$.
R_4 : $\mathsf{pt}(r, s)$:− $\mathsf{store}(p, q), \mathsf{pt}(p, r), \mathsf{pt}(q, s)$.

(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.

processing [25], networking [23], program analysis [6, 15], and robotics [36]. Datalog is an appealing target because it is expressive enough—it captures all PTIME problems [11]—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 propagation procedure that leverages provenance information that is naturally obtained from Datalog evaluation.

We evaluate DIFFLOG on a suite of 10 benchmarks from knowledge discovery and program analysis, and compare its performance and accuracy to two state-of-the-art systems: NEURALLP [47], which also leverages continuous reasoning, and METAGOL [9], which is based on discrete reasoning. DIFFLOG successfully learns the correct program on 8 of the 10 benchmarks, while METAGOL and NEURALLP are only able to handle 3 and 4 benchmarks respectively, and even then, learn a program with non-zero test error.

Illustrative Example. We illustrate DIFFLOG using the example of learning a popular program analysis called Andersen's pointer analysis [4], shown in Figure 1. This analysis reasons about the flow of information in pointer-manipulating programs at compile-time. It is central to statically detecting a broad range of software bugs, proving their absence, enabling advanced compiler optimizations, and so on [44]. For instance, consider the C program in Figure 1(a). It is evident that pointer `b1` points to (i.e., holds the address of) `c1` due to the statement `b1 = &c1`. However, it is not evident that `d` and `f` may also point to `c1` in some execution of 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 `b2` points to `c1`. 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) from the input/output data shown in Figure 1(c). The input data (also called extensional database or EDB) represents relevant facts about the input C program, such as tuples $\mathsf{addr}(x, y)$ for statements of the form x = &y, tuples $\mathsf{load}(x, y)$ for statements of the form x = *y, and so on. The output data (also called intentional database or IDB) represents the exact points-to information, namely, tuples $\mathsf{pt}(x, y)$ denoting that x may point to y. These are tuples corresponding 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_1, 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 possibly capture the exact points-to information for every C program. In this case, we should still learn rules that best approximates the training data. For instance, even though tuples `pt(b1, c2)` and `pt(b2, c1)` 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 [9] and ZAATAR [3] are severely limited in their ability to scale in the presence of recursion and increasing dataset size. As a result, METAGOL times out on 6 of the benchmarks on which DIFFLOG successfully learns the intended program.

2 Related Work

The most closely related to our work is Cohen et al. [7, 8, 47]. Their NEURALLP framework [47] was the first to propose a differentiable approach to learning the structure of logical rules. However, their approach targets a limited logic called TensorLog [7, 8] which does not support recursive rules, focuses on unary and binary relations, and only targets rules having the chain pattern. In contrast, DIFFLOG can support recursive rules, relations of arbitrary arity, and rules with richer patterns. Also, the underlying techniques for supporting efficient computation of gradients are fundamentally different: they leverage back-propagation in deep neural networks whereas we employ forward propagation based on provenance information. This difference makes DIFFLOG and NEURALLP complementary in their strengths: while NEURALLP can learn from very large datasets, but with only limited patterns, DIFFLOG requires the training dataset to be much smaller, but can learn complex patterns in this data.

Another recent differentiable approach is δILP [13]. It is capable of supporting restricted forms of recursion but is otherwise limited to binary relations, among other restrictions such as a fixed number of rules for each relation and a fixed number of literals per rule body. Moreover, they attach weights to each *combination* of rules, rather than to the rules directly. The number of these combinations grows exponentially as a function of the number of rules that define each relation. As a result, they only focus on problems where there are at most two rules that define each relation, thus limiting their expressive power.

The field of *inductive logic programming* (ILP) has extensively studied the problem of inferring logic programs from examples, as surveyed by Muggleton and De Raedt [28]. The literature on ILP spans a number of foundational theoretical concepts, including θ-subsumption [33], relative generalization [34], refinement [46], and others [12, 26, 27, 31]. Based on these theoretical results, a number of practical ILP systems have been developed [29, 37, 38, 48]. The most significant difference between our approach and ILP lies in the use of continuous versus discrete reasoning, which affects the scalability of rule learning and resilience to noise. Besides, ILP systems, whose goal is to primarily explain a phenomenon from vast amounts of mined data, are not adept at learning recursive rules that can only be inferred through *deep* inspection.

The field of program synthesis has targeted the problem of synthesizing recursive programs [2, 14, 20, 21, 32, 35]. Most of these works focus on recursive functional programs that manipulate

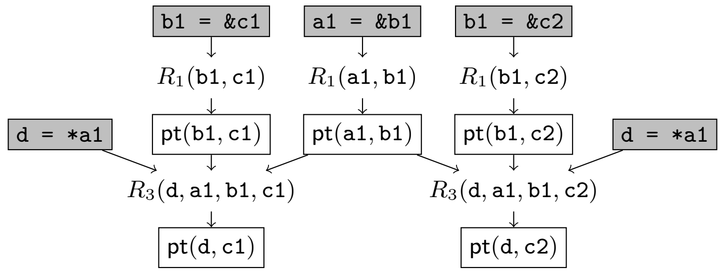


Figure 2: A portion of the derivation graph induced by applying the Datalog program of Figure 1 to the specified input relations. $R_1(\texttt{b1}, \texttt{c1})$ refers to the grounded constraint obtained by instantiating rule R_1 with $p = \texttt{b1}$ and $q = \texttt{c1}$, and similarly for all other rule instances.

recursive data structures. Datalog programs recursively traverse relations (hypergraphs). None of the functional techniques have been applied to this domain. ZAATAR [3] employs an SMT constraint solving approach to synthesize Datalog programs. It suffers from poor scalability because of discrete reasoning employed via the use of a theorem prover to search the space of Datalog programs and their derivation graphs.

3 Our Framework

3.1 Problem description

We begin with a brief overview of Datalog, as presented in [1]. We assume a collection of *relations*, $\{P, Q, \dots\}$. Each relation P has an arity k, and is a set of *tuples* $P(v_1, v_2, \dots, v_k)$, where v_1, v_2, …, v_k are *constants*. Examples include the relations $\texttt{pt}(p, q)$, $\texttt{addr}(p, q)$, 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:

$$P_h(\boldsymbol{u}_h) \coloneq P_1(\boldsymbol{u}_1), P_2(\boldsymbol{u}_2), \dots, P_k(\boldsymbol{u}_k),$$

where P_h is an output relation, and $\boldsymbol{u}_h$, $\boldsymbol{u}_1$, $\boldsymbol{u}_2$, …, $\boldsymbol{u}_k$ 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 = $*q$ ($\texttt{load}(p, q)$), and q may point to r ($\texttt{pt}(q, r)$), and r may point to s ($\texttt{pt}(r, s)$), then p may itself point to s ($\texttt{pt}(p, s)$)".

Instantiating a rule's variables yields a grounded constraint g of the form $P_1(\boldsymbol{v}_1) \wedge \cdots \wedge P_k(\boldsymbol{v}_k) \implies P_h(\boldsymbol{v}_h)$. In other words, given the set of antecedent tuples, $A_g = \{P_1(\boldsymbol{v}_1), P_2(\boldsymbol{v}_2), \dots, P_k(\boldsymbol{v}_k)\}$, the rule produces the conclusion $c_g = P_h(\boldsymbol{v}_h)$. 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: *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 $A_g = \{t_1, t_2, \dots, t_k\}$:

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

The central idea behind DIFFLOG is to switch the boolean operations $\vee$ and $\wedge$ 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:

$$v_t = \max_\tau (w_{g_1} \times w_{g_2} \times \cdots \times w_{g_n}), \tag{2}$$

where τ ranges over all derivation trees with conclusion t, and $g_1, g_2, \dots, g_n$ are the grounded constraints appearing in this tree, and where w_g is the weight of the associated rule for each grounded constraint g.

For example, the tuple $t_1 = \mathtt{pt}(\mathtt{b1}, \mathtt{c1})$ in Figure 2 is produced by one application of rule R_1, and the tuple $t_2 = \mathtt{pt}(\mathtt{d}, \mathtt{c1})$ 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 $w_{R_2} = 0.8$ 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 $(\vee, \wedge)$ with $(\max, \times)$ 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 [16]. Furthermore, when appropriately instrumented, classical algorithms to solve Datalog programs, such as the seminaive 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* 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 $\Pr(t_1 \wedge \neg t_2)$. 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 [10].

3.3 Learning DIFFLOG programs by numerical optimization

We evaluate DIFFLOG programs using a modified version of the seminaive algorithm for Datalog [1]. At a high level, at each time step $x \in \{0, 1, 2, \dots\}$, the evaluator maintains an association between output tuples t and their current candidate values v_t^x. The algorithm repeatedly considers instantiations g, all of whose antecedents $A_g = \{t_1, t_2, \dots, t_n\}$ satisfy $v_{t_i} \gneq 0$, and updates the candidate value for t:

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

To be able to compute the gradients ∇v_t with respect to the rule weights w_R, we also maintain a version of the *provenance polynomial* p_t for each tuple [16]. Informally, the provenance describes how the program concluded that t is an output tuple. We label each input tuple with the polynomial $p_t^0 = 1$, 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:

$$p_t^{x+1} = \begin{cases} p_t^x & \text{if } v_t^{x+1} = v_t^x, \text{ and} \\ w_g \times v_{t_1}^x \times v_{t_2}^x \times \cdots \times v_{t_n}^x & \text{otherwise,} \end{cases} \tag{4}$$

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, $f(\boldsymbol{w}) = \sum_t (v_t - l_t)^2$, 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 $w_R > w_0$, 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 $w_R \gneq 0$, 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 Section 4.1, this is important in improving generalization.

3.4 Implementation details

We now describe some additional details involved in implementing DIFFLOG. These involve: (*a*) the computation of the initial soup of candidate rules $\mathcal{R}$, and (*b*) optimizations to allow the DIFFLOG evaluation algorithm to reduce the time needed for each iteration of the optimizer.

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":

$$\begin{aligned}
&P_1(x, y) :\!-\ P_2(x, y), \\
&P_1(x, y) :\!-\ P_2(y, z), P_3(z, w), \text{ and} \\
&P_1(x, y) :\!-\ P_2(y, z), P_3(z, w), P_4(w, v),
\end{aligned}$$

Apart from the seminaive algorithm, the most significant 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, 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.

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 [47] and METAGOL [9]. 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

Table 1: Benchmark characteristics. The first five benchmarks are from the domain of knowledge discovery while the remaining five are from program analysis.

Benchmark	# Relations		# Rules		# Training tuples		# Test tuples	
	Input	Output	Expected	Candidates	Input	Output	Input	Output
Path	1	1	2	16	5	10	9	43
Ancestor	2	2	4	38	4	14	4	10
Animals	9	4	4	76	50	16	143	74
Samegen	1	1	2	154	7	21	7	18
Knights Move	4	1	1	40	930	210	210	42
Andersen	4	1	4	27	7	7	17	32
Escape	4	3	6	26	13	18	24	43
Modref	7	5	10	30	18	34	44	38
1-Call Site	7	2	4	94	28	16	67	138
Polysite	3	3	3	289	157	27	152	17

Benchmark	RMS Error on Test Set		
	DIFFLOG	NEURALLP	METAGOL
Path	0	N/A	timeout
Ancestor	0	0.45	timeout
Animals	0	N/A	0.53
Samegen	0	0.48	timeout
Knight Move	0	0	0
Andersen	0	N/A	timeout
Escape	0	0.39	timeout
Modref	0	N/A	timeout
1-Call Site	0.61	N/A	timeout
Polysite	0.31	N/A	0.23

Table 2: Test error achieved by DIFFLOG and the baseline learning algorithms on our benchmarks. N/A denotes "not applicable". Timeout is 6 hours.

Figure 3: Training and test RMSE as a function of number of selected rules from the DIFFLOG program.

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 Section 3.3. In Figure 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 RMS error achieved by each prefix of this ranked list. First, observe that optimum training and test errors is simultaneously achieved by approximately the same prefixes of the ranked list. Second, observe the long stretch of zero training error: in this region, for larger values of k, rules that did not produce any output tuples in the training dataset—and were consequently regarded by the optimization algorithm as harmless—start producing output tuples in the test set, thereby resulting in steadily increasing test error. Third, observe that, because we have not pruned the rules to only include "useful" tuples, test error does not drop to 0 at any point during the run. These observations demonstrate the importance of our reinterpretation heuristic from Section 3.3: we only keep those rules whose weights are above the given threshold, *and* which are useful in producing output tuples in the training dataset. As a sort of Occam's razor, this serves as a sort of regularization and prevents overfitting.

4.2 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.

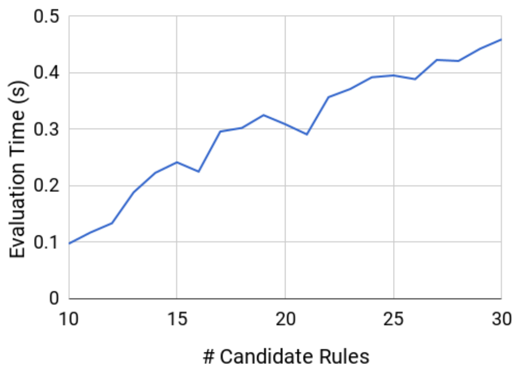


Figure 4: Evaluation time of DIFFLOG as a function of number of candidate rules for the Modref analysis.

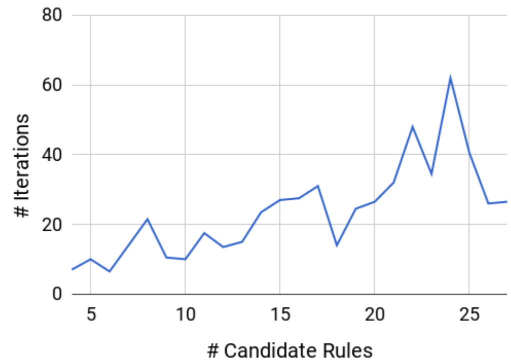


Figure 5: Number of optimization iterations required as a function of number of candidate rules for Andersen's analysis.

We present these results in Table 3. Observe that, in the presence of 1% noise, even though the training error is non-zero, the optimizer still learns the correct program, and achieves zero test error. In the presence of larger amounts of noise, the optimizer finds it increasingly difficult to fit the training data. However, in all these cases, the program it learns has low error on the uncorrupted test set. We therefore conclude that DIFFLOG is able to generalize even from moderately noisy data.

Table 3: Training and test error achieved by DIFFLOG on Andersen's analysis with noisy training data.

Proportion of Noise	**RMSE Train**	**RMSE Test**
0.01	0.089	0
0.05	0.16	0.083
0.10	0.294	0.166

4.3 Scalability of training process

Finally, we studied 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, and second, the number of iterations needed to converge to the final solution. We measured both quantities as a function of the number of candidate rules in $\mathcal{R}$. We present these plots in Figure 4 and 5 respectively. Observe that both quantities increase approximately linearly with the size of $\mathcal{R}$, suggesting that we can successfully learn complex programs from a large number of candidate rules.

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 such as gradient descent and Newton's method to synthesize Datalog programs. Our approach leverages provenance information that is naturally obtained from Datalog evaluation in order to efficiently forward-propagate the gradient through the rules to learn the weights. 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 negation. 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

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases: The logical level*. Pearson, 1st edition, 1994.

[2] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2013.

[3] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. Constraint-based synthesis of datalog programs. In *International Conference on Principles and Practice of Constraint Programming (CP)*, 2017.

[4] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, DIKU, University of Copenhagen, 1994. Ph.D. thesis.

[5] Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. Hinge-loss markov random fields and probabilistic soft logic. *Journal of Machine Learning Research*, 18, 2017.

[6] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2009.

[7] William Cohen. TensorLog: A differentiable deductive database. *CoRR*, abs/1605.06523, 2016. URL `http://arxiv.org/abs/1605.06523`.

[8] William Cohen, Fan Yang, and Kathryn Mazaitis. TensorLog: Deep learning meets probabilistic DBs. *CoRR*, abs/1707.05390, 2017. URL `http://arxiv.org/abs/1707.05390`.

[9] Andrew Cropper, Alireza Tamaddoni-Nezhad, and Stephen Muggleton. Meta-interpretive learning of data transformation programs. In *Inductive Logic Programming*. Springer, 2016.

[10] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 864–875. VLDB Endowment, 2004.

[11] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, September 2001.

[12] Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors. *Probabilistic Inductive Logic Programming: Theory and Applications*. Springer-Verlag, 2008.

[13] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, 2018. doi: 10.1613/jair.5714. URL `https://doi.org/10.1613/jair.5714`.

[14] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2016.

[15] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[16] Todd Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the 26th ACM Symposium on Principles of Database Systems*, PODS 2007, pages 31–40. ACM, 2007. ISBN 978-1-59593-685-1.

[17] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2), 2017.

[18] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the myria big data management service. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2014.

[19] Ross D. King. Applying inductive logic programming to predicting gene function. *AI Magazine*, 25(1), March 2004.

[20] Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7, 2006.

[21] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2013.

[22] Daphne Koller. Probabilistic relational models. In *9th International Workshop on Inductive Logic Programming*, 1999.

[23] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Communications of the ACM*, 52(11), November 2009.

[24] Brian Milch, Bhaskara Marthi, Stuart J. Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: probabilistic models with unknown objects. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

[25] Raymond J. Mooney. Inductive logic programming for natural language processing. In *6th International Workshop on Inductive Logic Programming*, 1996.

[26] Stephen Muggleton. Inverse entailment and Progol. *New generation computing*, 13(3-4), 1995.

[27] Stephen Muggleton and Wray L. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the International Conference on Machine Learning (ICML'88)*, 1988.

[28] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 1994.

[29] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *New Generation Computing*. Academic Press, 1990.

[30] Stephen Muggleton et al. Stochastic logic programs. *Advances in inductive logic programming*, 1996.

[31] Stephen H Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100 (1), 2015.

[32] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[33] Gordon D Plotkin. A note on inductive generalization. *Machine intelligence*, 5(1), 1970.

[34] Gordon D Plotkin. Automatic methods of inductive inference. Ph.D Thesis, Edinburgh University, 1972.

[35] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[36] David Poole. Logic programming for robot control. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[37] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3), September 1990.

[38] J. Ross Quinlan and R. Mike Cameron-Jones. Foil: A midterm report. In *Proceedings of the European Conference on Machine Learning (ECML'93)*, 1993.

[39] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.

[40] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62(1), Feb 2006.

[41] JCA Santos, H Nassif, D Page, SH Muggleton, and MJE Sternberg. Automated identification of protein-ligand interaction features using inductive logic programming: a hexose binding case study. *BMC Bioinformatics*, 13, 2012.

[42] Jiwon Seo, Stephen Guo, and Monica S Lam. SociaLite: Datalog extensions for efficient social network analysis. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2013.

[43] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with Datalog queries on Spark. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2016.

[44] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1), 2015.

[45] Leslie G. Valiant. Robust logics. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1999.

[46] Patrick R.J. van der Laag and Shan-Hwei Nienhuys-Cheng. Completeness and properness of refinement operators in inductive logic programming. *The Journal of Logic Programming*, 34 (3), 1998.

[47] Fan Yang, Zhilin Yang, and William Cohen. Differentiable learning of logical rules for knowledge base reasoning. In *Advances in Neural Information Processing Systems 30*, pages 2319–2328. 2017. URL `http://papers.nips.cc/paper/6826-differentiable-learning-of-logical-rules-for-knowledge-base-reasoning.pdf`.

[48] Qiang Zeng, Jignesh M. Patel, and David Page. Quickfoil: Scalable inductive logic programming. *Proceedings of the VLDB Endowment*, 8(3), November 2014.