Journal of Artificial Intelligence Research 67 (2020) 285-325 Submitted 05/2018; published 02/2020

TensorLog: A Probabilistic Database Implemented Using Deep-Learning Infrastructure

William W. Cohen WCOHEN@CS.CMU.EDU
Fan Yang FANYANG1@CS.CMU.EDU
Kathryn Rivard Mazaitis KRIVARD@CS.CMU.EDU
Machine Learning Department
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh PA 15208

Abstract

We present an implementation of a probabilistic first-order logic called TensorLog, in which classes of logical queries are compiled into differentiable functions in a neural-network infrastructure such as Tensorflow or Theano. This leads to a close integration of probabilistic logical reasoning with deep-learning infrastructure: in particular, it enables high-performance deep learning frameworks to be used for tuning the parameters of a probabilistic logic. The integration with these frameworks enables use of GPU-based parallel processors for inference and learning, making TensorLog the first highly parallellizable probabilistic logic. Experimental results show that TensorLog scales to problems involving hundreds of thousands of knowledge-base triples and tens of thousands of examples.

1. Introduction

Recent progress in deep learning has profoundly affected many areas of artificial intelligence. One exception to this trend is probabilistic first-order logical reasoning. In this paper, we seek to closely integrate probabilistic logic programs with the powerful infrastructures that have been developed for deep learning. The end goal is to enable deep learners to incorporate first-order probabilistic knowledge bases (KB), and conversely, to enable probabilistic reasoning over the outputs of deep learners.

Modern deep learning frameworks (Abadi et al., 2016; Bergstra et al., 2010) combine several useful properties. First, they allow researchers to easily formulate complex, expressive models for difficult tasks such as question-answering and image classification. Second, they support very fast execution of these models using parallelized operations on GPUs. Finally, by limiting the user to a library of matrix and tensor operations which are differentiable, they make learning very simple: one simply couples the model with an appropriate loss function, which measures how well the model fits training data, and then uses an off-the-shelf gradient-based optimizer to find parameter values that minimize loss. By using differentiable operations from a known library, algorithms for automatic reverse-mode differentiation (Pearlmutter & Siskind, 2008) can automatically compute the gradients needed for this optimization, so that implementing a learner for a specific model is reduced to the choice of the optimizer and any non-differentiable "hyperparameters". Most importantly, the use of uniform optimization schemes for learning encourages a reusability for learnable components—one can often build up complex models by combining well-understood submodules.

©2020 AI Access Foundation. All rights reserved.

```
answer(Question,Answer) :-
    classification(Question,aboutActedIn),
    mentionsEntity(Question,Entity), actedIn(Answer,Entity).
answer(Question,Answer) :-
    classification(Question,aboutDirected),
    mentionsEntity(Question,Entity), directed(Answer,Entity).
answer(Question,Answer) :-
    classification(Question,aboutProduced),
    mentionsEntity(Question,Entity), produced(Answer,Entity).
...
mentionsEntity(Question,Entity) :-
    containsNGram(Question,NGram), matches(NGram,Name),
    possibleName(Entity,Name), popular(Entity).

classification(Question,Y) :-
    containsNGram(Question,NGram), indicatesLabel(NGram,Y).
matches(NGram,Name) :-
    containsWord(NGram,Word), containsWord(Name,Word), important(Word).
```

Figure 1: A simple theory for question-answering against a KB.

While such deep learning frameworks are very useful, it is arguably clearer and more natural to write certain types of models using the language of logic—or more precisely, probabilistic first-order logic. For example, consider the program of Figure 1, which could be plausibly used for answering simple natural-language questions against a KB, such as "Who was the director of Apocalyse Now?" The main predicate `answer` takes a question and produces an answer (which would be an entity in the KB). The predicates `actedIn`, `directed`, etc, are from the KB. For the purpose of performing natural-language analysis, the KB has also been extended with facts about the text that composes the training and test data: the KB stores information about word n-grams contained in the question, the strings that are possible names of an entity, and the words that are contained in these names and n-grams. The underlined predicates `indicatesLabel`, `important`, and `popular` are "soft" KB predicates, and the goal of learning is to find appropriate weights for the soft-predicate facts—e.g., to learn that `indicatesLabel(director, aboutDirected)` has high weight. Ideally these weights would be learned indirectly, from observing inferences made using the KB. In this case we would like to learn from question-answer pairs, which rely indirectly on the KB predicates like `actedIn`, etc, rather than from hand-classified questions, or judgements about specific facts in the soft predicates.

Although probabilistic logics for theories like that of Figure 1 exist, they make use of special-purpose learners, and cannot be easily combined with or integrated with deep-learning components. TensorLog, the system we describe here, allows learning for probabilistic logics inside a deep-learning framework. This makes this possible to use logics inside a deep-learning model, or to use deep-learning subcomponents inside a logic. The efficiency

of the deep-learning frameworks also allows TensorLog learn using GPUs. Modern GPUs allow thousands of operations to be executed in parallel, which leads to gains in efficiency. For instance, for a variant of the problem above, we can learn from 10,000 questions against a KB of 420,000 tuples in around 200 seconds per epoch, on a typical desktop with a single GPU.

The main technical obstacle to integration of probabilistic logics into deep learners is that most existing first-order probabilistic logics are not easily adapted to evaluation on a GPU. One superficial problem is that the computations made in theorem-proving are not numeric, but there is also a more fundamental problem, which we will now discuss.

The most common approach to first-order inference is to "ground" a first-order logic by converting it to a zeroth-order format, such as a boolean formula or a probabilistic graphical model. For instance, in the context of a particular KB, the rule

$$p(X,Y) \leftarrow q(Y,Z), r(Z,Y) \tag{1}$$

can be "grounded" as the following finite boolean disjunction, where $\mathcal{C}$ is the set of objects in the KB:

$$\bigvee_{\exists x,y,z \in \mathcal{C}} (p(x,y) \vee \neg q(y,z) \vee \neg r(z,y))$$

This Boolean disjunction can be embedded in a neural network, e.g. to initialize an architecture (Towell, Shavlik, & Noordewier, 1990) or as a regularizer (Hu, Ma, Liu, Hovy, & Xing, 2016; Rocktäschel, Singh, & Riedel, 2015). For probabilistic first-order languages (e.g., Markov logic networks Richardson & Domingos, 2006), grounding typically results in a directed graphical model. This work is surveyed elsewhere (Kimmig, Mihalkova, & Getoor, 2015).

The problem with this approach is that groundings can be very large: even the small rule above gives a grounding of size $o(|\mathcal{C}|^3)$, which is likely much larger than the size of the KB, and a grounding of size $o(|\mathcal{C}|^n)$ is produced by a rule like

$$p(X_0, X_n) \leftarrow q_1(X_0, X_1), q_2(X_1, X_2), \ldots, q_n(X_{n-1}, X_n) \tag{2}$$

The target architecture for modern deep learners is based on GPUs, which have limited memory: hence the grounding approach can be used only for small KBs and short rules. For example, Serafini and d'Avila Garcez (2016) describe experimental results with five rules and a few dozen facts, and the largest datasets considered by Sourek, Aschenbrenner, Zelezný, and Kuzelka (2015) contain only about 3500 examples.

Although not all probabilistic logic implementations require explicit grounding, a similar problem arises in using neural-network infrastructures to implement any probabilistic logic which is computationally hard. For many probabilistic logics, answering queries is #P-complete or worse. Since the networks constructed in modern deep learning systems can be evaluated in time polynomial in their size, no polysize network can implement such a logic, unless #P=P, or inputs are very limited in size.

This paper addresses these obstacles with several interrelated contributions. First, in Section 2, we identify a restricted family of probabilistic deductive databases (PrDDBs) called *polytree-limited stochastic deductive knowledge graphs (ptree-SDKGs)* which are tractable, but still reasonably expressive. This formalism is a variant of stochastic logic

programs (SLPs). We also show that ptree-SDKGs are in some sense maximally expressive, in that we cannot drop the polytree restriction, or switch to a more conventional possible-worlds semantics, without making inference intractible.

Next, in Section 3, we present a novel algorithm for performing inference for ptree-SDKGs. This algorithm performs inference with a dynamic-programming method, which we formalize as belief propagation on a certain factor graph, where each random variable in the factor graph correspond to possible bindings to a logical variable in a proof, and the factors correspond to database predicates. In other words, the random variables are multinomials over all constants in the database, and the factors constrain these bindings to be consistent with database predicates that relate the corresponding logical variables. Although this is a simple idea, to our knowledge it is novel.

We also show that this inference process can be implemented under the constraints imposed by current neural-network infrastructures—i.e., that it can be implemented using matrix-vector operations which can be executed on a GPU. This property also distinguishes ptree-SDKGs from other logics, since usually logical inference for first-order expressions requires dynamic allocation of memory (to describe inferred facts), which cannot be easily performed using matrix operations. This property is practically important, since it means that inference can be compiled to deep learning frameworks such as Tensorflow, and integrated with other learning and optimization mechanisms. TensorLog can also be executed in parallel on GPUs, leading to significant speedups in inference and learning. Hence we also discuss in some detail our implementation of this logic, called TensorLog.

We then discuss related work, experimental results, and present conclusions.

To summarize, the main contributions of this paper are: (1) identification of a novel subset of first-order logic (ptree-SDKGs) which is maximally expressive, while retaining tractibility, along with the accompanying analysis; (2) an novel efficient inference algorithm for ptree-SDKGs; (3) discussion of a Tensorflow-based highly parallel implementation of the inference algorithm; and (4) experimental support for TensorLog's scalability and efficiency, and strong experimental results on one large-scale learning task.

2. Background

In this section we summarize the formal background needed for this paper with respect to deductive databases and stochastic logic programs, and present some complexity results for a restricted subset of stochastic logic programs.

2.1 Deductive DBs

In this subsection we review the usual definitions for logic programs and deductive databases, and also introduce the term *deductive knowledge graph (DKG)* for deductive databases containing only unary and binary predicates. This subsection can be omitted by readers familiar with logic programming.

An example of a *deductive database* (DDB) is shown in Figure 2. A *database*, $\mathcal{DB}$, is a set $\{f_1, \ldots, f_N\}$ of ground facts. (For the moment, ignore the numbers associated with each database fact in the figure.) A theory, $\mathcal{T}$, is a set of function-free Horn clauses. Clauses are written A:-$B_1, \ldots, B_k$, where A is called the *head* of the clause, $B_1, \ldots, B_k$ is the *body*, and A and the B_i's are called *literals*. Literals must be of the form $p(X_1, \ldots, X_k)$, where p is a

1. `uncle(X,Y):-child(X,W),brother(W,Y).`
2. `uncle(X,Y):-aunt(X,W),husband(W,Y).`
3. `status(X,tired):-child(W,X),infant(W).`

`child(liam,eve)`	0.99
`child(dave,eve)`	0.99
`child(liam,bob)`	0.75
`husband(eve,bob)`	0.9
`infant(liam)`	0.7
`infant(dave)`	0.1
`aunt(joe,eve)`	0.9
`brother(eve,chip)`	0.9

Figure 2: An example theory and database. Uppercase symbols are universally quantified variables, and so clause 3 should be read as a logical implication: for all database constants c_X and c_W, if `child(`c_X`,`c_W`)` and `infant(`c_W`)` can be proved, then `status(`c_X`,tired)` can also be proved.

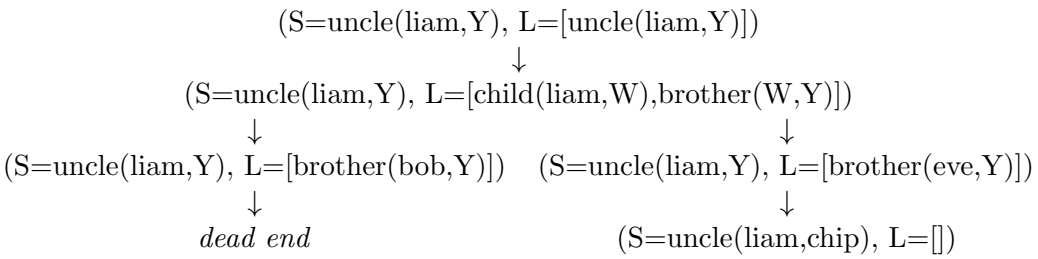

Figure 3: An example proof tree. From root to second level uses rule 1; next level uses unit clause `child(liam,bob):-` on left and unit clause `child(liam,eve):-` on right; final level uses `brother(eve,chip):-` on the right.

predicate symbol and the X_i's either logical variables or database constants. The set of all database constants is written $\mathcal{C}$. The number of arguments k to a literal is called its *arity*.

In this paper we focus on the case where all literals are binary or unary, i.e., have arity of no more than two. We will call such a database a *knowledge graph (KG)*, and the program a *deductive knowledge graph (DKG)*. We will also assume that constants appear only in the database, not in the theory (although this assumption can be relaxed).

Clauses can be understood as logical implications. Let σ be a *substitution*, i.e., a mapping from logical variables to constants in $\mathcal{C}$, and let $\sigma(L)$ be the result of replacing all logical variables X in the literal L with $\sigma(X)$. A set of tuples S is *deductively closed* with respect to the clause $A \leftarrow B_1, \ldots, B_k$ iff for all substitutions σ, either $\sigma(A) \in S$ or $\exists B_i : \sigma(B_i) \notin S$. For example, if S contains the facts of Figure 2, S is not deductively closed with respect to the clause 1 unless it also contains `uncle(chip,liam)` and `uncle(chip,dave)`. The *least model* for a pair $\mathcal{DB}, \mathcal{T}$, written *Model*$(\mathcal{DB}, \mathcal{T})$, is the smallest superset of $\mathcal{DB}$ that is deductively closed with respect to every clause in $\mathcal{T}$. This least model is unique, and in the usual DDB semantics, a ground fact f is considered "true" iff $f \in$ *Model*$(\mathcal{DB}, T)$.

There are two broad classes of algorithms for inference in a DDB. *Bottom-up inference* explicitly computes the set *Model*$(\mathcal{DB}, \mathcal{T})$ iteratively. Bottom-up inference repeatedly extends a set of facts S, which initially contains just the database facts, by looking for rules which "fire" on S and using them to derive new facts. (More formally, one looks for rules $A \leftarrow B_1, \ldots, B_k$ and substitutions σ such that $\forall i$, $\sigma(B_i) \in S$, and then adds the derived fact $\sigma(A)$ to S.) This process is then repeated until it converges. For DDB programs, bottom-up inference takes time polynomial in the size of the database $|\mathcal{DB}|$, but exponential in the length of the longest clause in $\mathcal{T}$ (Ramakrishnan & Ullman, 1995).

One problem with bottom-up theorem-proving is that it explicitly generates *Model*$(\mathcal{DB}, \mathcal{T})$, which can be much larger than the original database. The alternative is *top-down inference*. Here, the algorithm does not compute a least model explicitly: instead, it takes as input a query fact f and determines whether f is derivable, i.e., if $f \in$ *Model*$(\mathcal{DB}, \mathcal{T})$. More generally, one might retrieve all derivable facts that match some pattern, e.g., find all values of Y such that `uncle(joe,Y)` holds. (Formally, given $Q =$ `uncle(joe,Y)`, we would like to find all $f \in$ *Model*$(\mathcal{DB}, \mathcal{T})$ which are *instances of* Q, where an f is defined to be an *instance of* Q iff $\exists \sigma : f = \sigma(Q)$..

To describe top-down theorem-proving, we note that facts in the database can also be viewed as clauses: in particular a fact $p(a, b)$ can be viewed as a clause $p(a, b) \leftarrow$ which has $p(a, b)$ as its head and an empty body. This sort of clause is called a *unit clause*. We will use $\mathcal{T}^{+\mathcal{DB}}$ to denote the theory $\mathcal{T}$ augmented with unit clauses for each database fact. A top-down theorem prover can be viewed as constructing and searching a following tree, using the theory $\mathcal{T}^{+\mathcal{DB}}$. The process is illustrated in Figure 3, and detailed below.

1. The root vertex is a pair (S, L), where S is the query Q, and L is a list containing only Q. In general every vertex is a pair where S is something derived from Q, and L is a list of literals left to prove.

2. For any vertex (S, L), where $L = [G_1, \ldots, G_n]$, there is a child vertex (S', L') for each rule $A \leftarrow B_1, \ldots, B_k \in \mathcal{T}^{+\mathcal{DB}}$ and substitution σ for which $\sigma(G_i) = \sigma(A)$ for some G_i. In this child node, $S' = \sigma(S)$, and

$$L' = [\sigma(G_1), \ldots, \sigma(G_{i-1}), \sigma(B_1), \ldots, \sigma(B_k), \sigma(G_{i+1}), \ldots, \sigma(G_n)]$$

Note that L' is smaller than L if the clause selected is a unit clause (i.e., a fact). If L' is empty, then the vertex is called a *solution vertex*. In any solution vertex (S, L), if S contains no variables,[1] then S is an instance of Q and is in *Model*$(\mathcal{T}, \mathcal{DB})$.

If $\mathcal{T}$ is not recursive, or if recursion is limited to a fixed depth, then the proof graph is finite. We will restrict our discussion below to theories with finite proof graphs. For this case, the set of all answers to a query Q can be found by systematically searching the proof tree for all solution vertices. A number of strategies exist for this, but one popular one is that used by Prolog, which uses depth-first search, ordering edges by picking the first rule $A \leftarrow B_1, \ldots, B_k$ in a fixed order, and only matching rules against the first element of L. This strategy can be implemented quite efficiently and is easily extended to much more general logic programs.

1. If S does have variables in it, then any fact f which can be constructed by replacing variables in Q with database constants is in the least model. For clarity we will ignore this complication in the discussion below.

2.2 Stochastic Logic Programs

There are a number of approaches to incorporating probabilistic reasoning in first-order logics. TensorLog's semantics are based on *stochastic logic programs (SLPs)* (Cussens, 2001), in which the theory $\mathcal{T}$ is extended by associating with each rule r a non-negative scalar weight θ_r. Below we summarize the semantics associated with SLPs, for completeness, and refer the reader to Cussens (2001) for details.

In an SLP weights θ_r are added to edges of the top-down proof graph the natural way: when a rule r is used to create an edge $(S, L) \rightarrow (S', L')'$, this edge is given weight θ_r. We define the weight of a path $v_1 \rightarrow \ldots \rightarrow v_n$ in the proof graph for Q to be the product of the weights of the edges in the path, and the weight of a node v to be the sum of the weights of the paths from the root note $v_0 = (Q, [Q])$ to v. If $r_{v,v'}$ is the rule used for the edge from v to v', then the weight of $w_Q(v_n)$ is

$$w_Q(v_n) \equiv \sum_{v_0 \rightarrow \ldots \rightarrow v_n} \prod_{i=0}^{n-1} \theta_{r_{v_i,v_{i+1}}}$$

The weight of an answer f to query Q is defined by summing over paths to solution nodes that yield f. Let support(f) be the set of all graph nodes (S, L) where $S = f$ and L is an empty list, and define

$$w_Q(f) \equiv \sum_{v \in \text{support}(f)} w_Q(v) \tag{3}$$

Finally, if we assume that some answers to Q do exist, we can produce a conditional probability distribution over answers f to the query Q by normalizing w_Q, i.e.,

$$\Pr(f|Q) \equiv \frac{1}{Z} w_Q(f) \tag{4}$$

where $Z = \sum_f w_Q(f)$.

Following the terminology of (Cussens, 2001) this is a *pure unnormalized SLP*. SLPs were originally defined (Muggleton et al., 1996) for a fairly expressive class of logic programs, namely all programs which are *fail free*, in the sense that there are no "dead ends" in the proof graph (i.e., from every vertex v, at least one solution node is reachable). Prior work with SLPs also considered the special case of *normalized SLPs*, in which the weights of all outgoing edges from every vertex v sum to one. For normalized fail-free SLPs, it is simple to modify the usual top-down theorem prover to sample from $Pr(f|Q)$.

SLPs are closely connected to several other well-known types of probabilistic reasoners. SLPs are defined by introducing probabilistic choices into a top-down theorem-proving process: since top-down theorem-proving for logic programs is analogous to program execution in ordinary programs, SLPs can be thought of as logic-program analogs to probabilistic programming languages like Church (Goodman, Mansinghka, Roy, Bonawitz, & Tenenbaum, 2012). Normalized SLPs are also conceptually quite similar to stochastic grammars, such as PCFGS, except that stochastic choices are made during theorem-proving, rather than rewriting a string.

2.3 Stochastic Deductive KGS: a Tractible Restriction of SLPs

The logic used in TensorLog is based on SLPs, but includes several important restrictions, which together make inference tractible. First, we restrict the program to be in DDB form—i.e., it consists of a theory $\mathcal{T}$ which contains function-free clauses, and a database $\mathcal{DB}$ (of unit clauses). Second, we restrict all predicates to be unary or binary. Third, we restrict the clauses in the theory $\mathcal{T}$ to have weight 1, so that the only meaningful weights are associated with database facts. We call this restricted SLP a *stochastic deductive knowledge graph (SDKG)*. Finally, we will impose an additional syntactic restriction on SDKGs, to be discussed below, by requiring them to be "polytree-limited".

The first three restrictions simply combine the SLP semantics with a restrictions associated with a deductive knowledge graph. One might hope that there restrictions would make inference tractible, but this is not the case: even for very simple SDKGs, computing $P(f|Q)$ is #P-hard.

Theorem 1 *Computing $P(f|Q)$ (relative to a SDKG $\mathcal{T}, \mathcal{DB}$) for all possible answers f of the query Q is #P-hard, even if there are only two such answers, the theory contains only two non-recursive[2] clauses, and the KG contains only 13 facts.*

A proof appears in the appendix. While this result may seem surprising, we note that it is easy to find small theories with exponentially many proofs: e.g., the clause of Equation2 can have exponentially many proofs, and straightforward adaptations of non-probabilistic theorem proving methods are expensive on such clauses.

Fortunately, however, one further restriction makes SDKG inference tractible. For a theory clause $r = A \leftarrow B_1, \ldots, B_k$, define the *literal influence graph for r* to be a graph where each B_i is a vertex, and there is an edge from B_i to B_j iff they share a variable. A graph is a *polytree* iff there is at most one path between any pair of vertices: i.e., if each strongly connected component of the graph is a tree. Finally, we define a theory to be *polytree-limited* iff the influence graph for every clause is a polytree. Figure 4 contains some examples of polytree-limited clauses. The theorem below shows that this additional restriction makes inference tractable.

Theorem 2 *For any SDKG with a non-recursive polytree-limited theory $\mathcal{T}$, $P(f|Q)$ can be computed in time linear in the size of $\mathcal{T}$ and $\mathcal{DB}$.*

The proof follows from the correctness of a dynamic-programming algorithm for SDKG inference, which we will present below in Section 3. In brief, the algorithm is based on belief propagation in a certain factor graph. We construct a graph where the random variables are multinomials over the set of all database constants, and each random variable corresponds to a logical variable in the proof graph. The logical literals in a proof correspond to factors, which constrain the bindings of the variables to make the literals true. Importantly for the goal of compilation into deep-learning frameworks, the message-passing steps used for belief propagation can be defined as numerical operations, and given a predicate and an input/output mode, the message-passing steps required to perform belief propagation (and hence inference) can be "unrolled" into a function, which is differentiable.

2. A theory is *recursive* if some proof of a query $q(x_1, y_1)$ involves a subgoal of the form $q(x_2, y_2)$, where the x's and y's are either constants or variables.

2.4 Complexity of Stochastic DKGs Variants

Below we will discuss some of the potential ways in which polytree-limited SDKGs might be plausibly extended.

As one extension, we note that constants can be allowed the theory while maintaining tractibility. Above we assume that constants appear only in the database, not in the theory. However, it is possible to introduce a constant into a theory by creating a special unary predicate which holds only for that constant: e.g., to use the constant `tired`, one could create a database predicate `assign_tired(T)` which contains the one fact `assign_tired(tired)`, and use it to introduce a variable which is bound to the constant `tired` when needed. For instance, the clause 3 of Figure 2 would be rewritten as

```
status(X,T):-assign_tired(T),child(X,W),infant(W).        (5)
```

where $\mathcal{DB}$ conceptually contains one `assign_tired` fact, `assign_tired(tired)`. In the current TensorLog implementation, we allow constants to appear in theories, but only in literals of this sort.

One can also allow weights on rules from the theory, rather than only on *facts* in the databases. The usual "trick" to lift weights from a database into rules is to introduce a special clause-specific fact, and add it to the clause body (Poole, 1997). For example, a weighted version of clause 3 could be re-written as

```
status(X,tired):-assign_c3(RuleId),weighted(RuleId),child(W,X),infant(W)
```

where the (parameterized) fact `weighted(c3)` appears in $\mathcal{DB}$, and the constant `c3` appears nowhere else in $\mathcal{DB}$. This same trick can be extended to allow even more expressive rule-weighting schemes: in particular, ProPPR (Wang, Mazaitis, & Cohen, 2013) allows one to attach a computed set of features to a rule in order to weight it: e.g., one can write

```
status(X,tired):-{weighted(A):child(W,X),age(W,A)}
```

which indicates that all the ages of the children of X should be used as features to determine if the rule succeeds. This is equivalent to the rule `status(X,tired) :-child(W,X), age(W,A), weighted(A)`. TensorLog allows these sorts of computed weighted features to be attached to a rule using the ProPPR syntax, and automatically converts to using the construction above. In the experiments below, we use this construction for experimental comparisons to ProPPR.

We finally discuss one arguably desirable extension which cannot be implemented without losing tractibility: an extension to *possible-worlds semantics*. In the SLP semantics, the parameters Θ only have meaning in the context of the set of proofs derivable using the theory $\mathcal{T}$. This can be thought of as a "possible proofs" semantics. It has been argued that it is more natural to adopt a "possible worlds" semantics, in which Θ is used to define a distribution, $\Pr(I|\mathcal{DB},\Theta)$, over "hard" databases, and the probability of a derived fact f is defined as follows, where $[\![\cdot]\!]$ is a zero-one indicator function:

$$\Pr_{\texttt{TupInd}}(f|\mathcal{T},\mathcal{DB},\Theta) \equiv \sum_I [\![f \in \mathit{Model}(I,\mathcal{T})]\!] \cdot \Pr(I|\mathcal{DB},\Theta) \tag{6}$$

Potential hard databases are often called *interpretations* in this setting. The simplest such "possible worlds" model is the *tuple independence* model for PrDDB's (Suciu, Olteanu, Ré, & Koch, 2011): in this model, to generate an interpretation I, each fact $f \in \mathcal{DB}$ is sampled by independent coin tosses, i.e., $\Pr_{\texttt{TupInd}}(I|\mathcal{DB}, \Theta) \equiv \prod_{t \in I} \theta_t \cdot \prod_{t \in \mathcal{DB}-I}(1-\theta_t)$.

ProbLog (Fierens, Van den Broeck, Renkens, Shterionov, Gutmann, Thon, Janssens, & De Raedt, 2015) is one well-known logic programming language which adopts this semantics, and there is a large literature on approaches to more tractibly estimating Eq 6, which naively requires marginalizing over all $2^{|\mathcal{DB}|}$ interpretations (Suciu et al., 2011; De Raedt & Kersting, 2008). A natural question to ask is whether polytree-limited SDKGs, which are tractible under the possible-proofs semantics of SLPs, are also tractible under a possible-worlds semantics. Unfortunately, this is not the case.

Theorem 3 *Computing $P(f)$ in the tuple-independent possible-worlds semantics for a single ground fact f is #P-hard.*

This result is well known (Suciu et al., 2011) but for completeness the appendix to this paper contains a proof, which emphasizes the fact that reasonable syntactic restrictions (such as polytree-limited theories) are unlikely to make inference tractible. In particular, the theory used in the construction is extremely simple: all predicates are unary, and contain only three literals in their body.

3. Efficient Differentiable Inference for Polytree-Limited SDKGs

In this section we present an efficient dynamic-programming method for inference in polytree-limited SDKGs. We formalize this method as belief propagation on a certain factor graph, where the random variables in the factor graph correspond to possible bindings to a logical variable in a proof, and the factors correspond to database predicates. In other words, the random variables are multinomials over all constants in the database, and the factors will constrain these bindings to be consistent with database predicates that related the corresponding logical variables.

Although using belief propagation in this way is a simple idea, to our knowledge it is a novel method for first-order probabilistic inference. Certainly it is quite different from more common formulations of first-order probabilistic inference, where random variables typically are Bernoulli random variables, which correspond to *potential* ground database facts (i.e., elements of the Herbrand base of the program).

Although the inference scheme we describe is equivalent to belief propagation, because our ultimate goal is integration with neural network infrastructures, we will implement an "unrolled" version of this belief propagation in terms of a series of numeric functions, each of which finds answers to a particular family of queries. The details of this are discussed below.

3.1 Numeric Encoding of PrDDB's and Queries

It will be convenient to encode the database numerically. We will assume all constants have been mapped to integers. For a constant $c \in \mathcal{C}$, we define $\mathbf{u}_c$ to be a one-hot row-vector representation for c, i.e., a row vector of dimension $|\mathcal{C}|$ where $\mathbf{u}[c] = 1$ and $\mathbf{u}[c'] = 0$

for $c' \neq c$. We can also represent a binary predicate p by a sparse matrix $\mathbf{M}_p$, where $\mathbf{M}_p[a,b] = \theta_{p(a,b)}$ if $p(a,b) \in \mathcal{DB}$, and a unary predicate q as an analogous row vector $\mathbf{v}_q$. Notice that $\mathbf{M}_p$ encodes information not only about the database facts in predicate p, but also about their parameter values; also, note that $\mathbf{M}_p$ is sparse, and does not contain weights for facts $p(x,y)$ which do not appear in the database. Collectively, the matrices $\mathbf{M}_{p_1}, \ldots, \mathbf{M}_{p_n}$ for the predicates $p_1, \ldots, p_n$ can be viewed as a three-dimensional tensor.

Our main interest here is queries that retrieve all derivable facts that match some query Q: e.g., to find all values of Y such that `uncle(joe,Y)` holds. We define an *argument-retrieval query* Q as query of the form $p(c,Y)$ or $p(Y,c)$. We say that $p(c,Y)$ has an *input-output mode* of `in,out` and $p(Y,c)$ has an input-output mode of `out,in`. For the sake of brevity, below we will assume below the mode `in,out` when possible, and abbreviate the two modes as `io` and `oi`.

The *response* to a query $p(c,Y)$ is a distribution over possible substitutions for Y, encoded as a vector $\mathbf{v}_Y$ such that for all constants $d \in \mathcal{C}$, $\mathbf{v}_Y[d] = \Pr(p(c,d)|Q = p(x,Y), \mathcal{T}, \mathcal{DB}, \Theta)$. Note that in the SLP model $\mathbf{v}_Y$ is a conditional probability vector, conditioned of $Q = p(c,Y)$, which we will sometimes emphasize with denoting it as $\mathbf{v}_{Y|c}$. Formally if $U_{p(c,Y)}$ encodes the set of facts f that "match" (are instances of) $p(c,Y)$, then

$$\mathbf{v}_{Y|c}[d] = \Pr(f = p(c,d)|f \in U_{p(c,Y)}, \mathcal{T}, \mathcal{DB}, \Theta) \equiv \frac{1}{Z} w_Q(f = p(c,d))$$

Although here we only consider single-literal queries, we note that more complex queries can be answered by extending the theory: e.g., to find

$$\{\texttt{Y: uncle(joe,X),husband(X,Y)}\}$$

we could add the clause `q1(Y):-uncle(joe,X),husband(X,Y)` to the theory and find the answer to `q1(Y)`.

Since the goal of our reasoning system is to correctly answer queries using functions, we also introduce a notation for functions that answer particular types of queries: in particular, for a predicate symbol p, $f^p_{\texttt{io}}$ denotes a *query response function* for all queries with predicate p and mode `io`. We define a *query response function* for a query of the form $p(c,Y)$ to be a function which, when given a one-hot encoding of c, $f^p_{\texttt{io}}$ returns the appropriate conditional probability vector:

$$f^p_{\texttt{io}}(\mathbf{u}_c) \equiv \mathbf{v}_{Y|c} \tag{7}$$

We analogously define $f^p_{\texttt{oi}}$. Finally, we define $g^p_{\texttt{io}}$ to be the unnormalized version of this function, i.e., the weight of f according to $w_Q(f)$:

$$g^p_{\texttt{io}}(\mathbf{u}_c) \equiv w_Q(f)$$

For convenience, we will introduce another special DB predicate `any`, where `any`(a,b) is conceptually true for any pair of constants a,b; however, as we show below, the matrix $\mathbf{M}_{\texttt{any}}$ need not be explicitly stored. We also constrain clause heads to contain distinct variables which all appear also in the body.

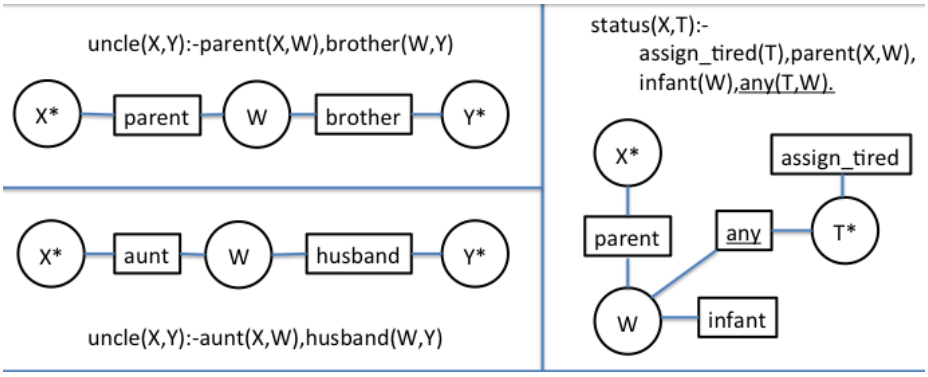

Figure 4: Examples of factor graphs for the example theory.

3.2 Efficient Inference for One-Clause Theories

As noted above, Θ can be used to encode confidences on either facts or rules (by using clause-specific facts, as described in Section 2.4). As defined above, f^p and g^p are numeric functions which depend on Θ. Below we will outline a scheme for implementing these functions which makes them not only numeric, but also differentiable: that is, we will define them so that they vary smoothly as the weights in Θ change. Differentiability of these functions will make it possible to use gradient-based methods to optimize Θ, and hence allow weights to be learned.

We will start by considering a highly restricted class of theories $\mathcal{T}$, namely programs containing only one non-recursive polytree-limited clause r that obeys the restrictions above. We build a factor graph G_r for r as follows: for each logical variable W in the body, there is a random variable W; and for every literal $q(W_i, W_j)$ in the body of the clause, there is a factor with potentials $\mathbf{M}_q$ linking variables W_i and W_j. Finally, if the factor graph is disconnected, we add `any` factors between the components until it is connected.[3]

Figure 4 gives examples. The variables appearing in the clause's head are starred. The correctness of this procedure follows immediately from the convergence of belief propagation on factor graphs for polytrees (Kschischang, Frey, & Loeliger, 2001).

BP over G_r can now be used to compute the conditional vectors $f^p_{\texttt{io}}(\mathbf{u}_c)$ and $f^p_{\texttt{oi}}(\mathbf{u}_c)$. For example to compute $f^p_{\texttt{io}}(\mathbf{u}_c)$ for clause 1, we would set the message for the evidence variable X to $\mathbf{u}_c$, run BP, and read out as the value of f the marginal distribution for Y.

3.3 Differentiable Inference for One-Clause Theories

To make the final step toward integration of this algorithm with neural-network infrastructures, we must finally compute an explicit, differentiable, query response function, which computes $f^p_{\texttt{io}}(\mathbf{u}_c)$. To do this we "unroll" the message-passing steps into a series of operations. Figure 5 shows the algorithm used in the current implementation of TensorLog, which follows previous work in translating belief propagation to differentiable form (Gorm-

3. Any pair of variables in different chains can be connected together, since the `any` predicate does not constrain their joint binding. This predicate simply lets us construct a message that multiplies the weight of all solutions for the chain containing the input with the total weight of all solutions for a second chain.

define compileMessage($L \rightarrow X$):
  assume wolg that $L = q(X)$ or $L = p(X_i, X_o)$
  generate a new variable name $\mathbf{v}_{L,X}$
  if $L = q(X)$ **then**
    emitOperation($\mathbf{v}_{L,X} = \mathbf{v}_q$)
  else if X is the output variable X_o of L **then**
    $\mathbf{v}_i$ = compileMessage($X_i \rightarrow L$)
    emitOperation($\mathbf{v}_{L,X} = \mathbf{v}_i \cdot \mathbf{M}_p$)
  else if X is the input variable X_i of L **then**
    $\mathbf{v}_o$ = compileMessage($X_i \rightarrow L$)
    emitOperation($\mathbf{v}_{L,X} = \mathbf{v}_o \cdot \mathbf{M}_p^T$)
  return $\mathbf{v}_{L,X}$

define compileMessage($X \rightarrow L$):
  if X is the input variable X **then**
    return $\mathbf{u}_c$, the input
  else
    generate a new variable name $\mathbf{v}_X$
    assume $L_1, L_2, \ldots, L_k$ are the
      neighbors of X excluding L
    for $i = 1, \ldots, k$ **do**
      $\mathbf{v}_i$ = compileMessage($L_i \rightarrow X$)
    emitOperation($\mathbf{v}_X = \mathbf{v}_1 \circ \cdots \circ \mathbf{v}_k$)
    return $\mathbf{v}_X$

Figure 5: Algorithm for unrolling belief propagation on a polytree into a sequence of message-computation operations. Notes: (1) if $L = p(X_o, X_i)$ then replace $\mathbf{M}_p$ with $\mathbf{M}_p^T$ (the transpose). (2) Here $\mathbf{v}_1 \circ \mathbf{v}_2$ denotes the Hadamard (component-wise) product, and if $k = 0$ an all-ones vector is returned.

ley, Dredze, & Eisner, 2015). If a clause is not a polytree, then this is detected (by analyzing the literal influence graph for loops) and reported to the user.

In the code, we found it convenient to extend the notion of input-output modes for a query, as follows: a variable X appearing in a literal $L = p(X, Y)$ in a clause body is a *nominal input* if it appears in the input position of the head, or any literal to the left of L in the body, and is an *nomimal output* otherwise. In Prolog a convention is that nominal inputs appear as the first argument of a predicate, and in TensorLog, if the user respects this convention, then "forward" message-passing steps use $\mathbf{M}_p$ rather than $\mathbf{M}_p^T$ (reducing the cost of transposing large $\mathcal{DB}$-derived matrices, since our message-passing schedule tries to maximize forward messages.) The code contains two mutually recursive routines, and is invoked by requesting a message from the output variable to a fictional output literal. The result will be to emit a series of operations, and return the name of a register that contains the unnormalized conditional probability vector for the output variable. For instance, for the sample clauses, the functions returned are shown in Table 1.

Here we use $g^r_{\mathtt{io}}(\mathbf{u}_c)$ for the unnormalized version of the query response function build from G_r. One could normalize[4] as

$$f^p_{\mathtt{io}}(\mathbf{u}_c) \equiv g^r_{\mathtt{io}}(\mathbf{u}_c) / \|g^r_{\mathtt{io}}(\mathbf{u}_c)\|_1$$

4. Neither of these normalization schemes is well-defined when $g^r_{\mathtt{io}}(\mathbf{u}_c) = \mathbf{0}$, which is the result for a query that is not proveable at all. To avoid this, before normalization a small weight is given to a special "not proveable" constant. In learning, the "not proveable" constant is considered an incorrect answer to every query.

Rule	r1: uncle(X,Y):- parent(X,W), brother(W,Y)	r2: uncle(X,Y):- aunt(X,W), husband(W,Y)	r3: status(X,T):- assign_tired(T), parent(X,W), infant(W),any(T,W)
Function	$g^{r1}_{\texttt{io}}(\mathbf{u}_c)$	$g^{r2}_{\texttt{io}}(\mathbf{u}_c)$	$g^{r3}_{\texttt{io}}(\mathbf{u}_c)$
Operation sequence defining function	$\mathbf{v}_{1,W} = \mathbf{u}_c\mathbf{M}_{\texttt{parent}}$ $\mathbf{v}_W = \mathbf{v}_{1,W}$ $\mathbf{v}_{2,Y} = \mathbf{v}_W\mathbf{M}_{\texttt{brother}}$ $\mathbf{v}_Y = \mathbf{v}_{2,Y}$	$\mathbf{v}_{1,W} = \mathbf{u}_c\mathbf{M}_{\texttt{aunt}}$ $\mathbf{v}_W = \mathbf{v}_{1,W}$ $\mathbf{v}_{2,Y} = \mathbf{v}_W\mathbf{M}_{\texttt{husband}}$ $\mathbf{v}_Y = \mathbf{v}_{2,Y}$	$\mathbf{v}_{2,W} = \mathbf{u}_c\mathbf{M}_{\texttt{parent}}$ $\mathbf{v}_{3,W} = \mathbf{v}_{\texttt{infant}}$ $\mathbf{W} = \mathbf{v}_{2,W} \circ \mathbf{v}_{3,W}$ $\mathbf{v}_{1,T} = \mathbf{v}_{\texttt{assign\_tired}}$ $\mathbf{v}_{4,T} = \mathbf{v}_W\mathbf{M}_{\texttt{any}}$ $\mathbf{T} = \mathbf{v}_{1,T} \circ \mathbf{v}_{4,T}$
Returns	$\mathbf{v}_Y$	$\mathbf{v}_Y$	$\mathbf{v}_T$

Table 1: Chains of messages constructed for the three sample clauses shown in Figure 4, written as functions in pseudo code.

where r is the one-clause theory defining p. To make optimization easier in learning, however, TensorLog uses a "softmax" normalization:

$$f^p_{\texttt{io}}(\mathbf{u}_c) \equiv \text{softmax}(g^r_{\texttt{io}}(\mathbf{u}_c)) \tag{8}$$

where softmax($\mathbf{v}$) is a vector $\mathbf{s}$ so that $\mathbf{s}[i] = \exp(\mathbf{v}[i])/(\sum_{i'} \exp(\mathbf{v}[i']))$.

3.4 Differentiable Inference for Multi-Clause Programs

We now extend this idea to theories with many clauses. We first note that if there are several clauses with the same predicate symbol in the head, we simply sum the unnormalized query response functions: e.g., for the predicate `uncle`, defined by rules r_1 and r_2, we would define

$$g^{\texttt{uncle}}_{\texttt{io}} = g^{r1}_{\texttt{io}} + g^{r2}_{\texttt{io}}$$

This is equivalent to building a new factor graph G, which would be approximately $\cup_i G_{ri}$, together global input and output variables, plus a factor that constrains the input variables of the G_{ri}'s to be equal, plus a factor that constrains the output variable of G to be the sum of the outputs of the G_{ri}'s.

A more complex situation is when the clauses for one predicate, p, use a second theory predicate q, in their body: for example, this would be the case if `aunt` was also defined in the theory, rather than the database. For a theory with no recursion, we can replace the message-passing operations $\mathbf{v}_Y = \mathbf{v}_X\mathbf{M}_q$ with the function call $\mathbf{v}_Y = g^q_{\texttt{io}}(\mathbf{v}_X)$, and likewise the operation $\mathbf{v}_Y = \mathbf{v}_X\mathbf{M}^T_q$ with the function call $\mathbf{v}_Y = g^q_{\texttt{oi}}(\mathbf{v}_X)$. It can be shown that this is equivalent to taking the factor graph for q and "splicing" it into the graph for p.

It is also possible to allow function calls to recurse to a fixed maximum depth: we must simply add an extra argument that tracks depth to the recursively-invoked g^q functions, and make sure that g^p returns an all-zeros vector (indicating no more proofs can be found) when the depth bound is exceeded. Currently this is implemented by marking learned functions

g with the predicate q, a mode, and a depth argument d, and ensuring that function calls inside $g^p_{\mathtt{io},d}$ to q always call the next-deeper version of the function for q, e.g., $g^q_{\mathtt{io},d+1}$. The depth bound is associated with the compilation routine, and a non-default value can be provided by the user.

Computationally, the algorithm we describe is quite efficient. Assuming the matrices $\mathbf{M}_p$ exist, the additional memory needed for the factor-graph G_r is linear in the size of the clause r, and hence the compilation to response functions is linear in the theory size and the number of steps of BP. For ptree-SDKGs, when G_r is a tree, the number of message-passing steps is also linear. Message size is (by design) limited to $|\mathcal{C}|$, and is often smaller in practice, due to sparsity or type restrictions (discussed below).

3.5 Implementation: TensorLog

Compilation. The current implementation of TensorLog operates by first "unrolling" the belief-propagation inference to an intermediate form consisting of sequences of abstract operators, as suggested by the examples of Table 1. The "unrolling" code performs a number of optimizations to the sequence in-line: one important one is to use the fact that $\mathbf{v}_X \circ (\mathbf{v}_Y \mathbf{M}_{\mathtt{any}}) = \mathbf{v}_X \|\mathbf{v}_Y\|_1$ to avoid explicitly building $\mathbf{M}_{\mathtt{any}}$. (If a clause is not a polytree, this is detected when it is "unrolled", as at least one random variable will have two paths connecting it to the input, and an error is generated.) These abstract operator sequences are then "cross-compiled" into expressions on one of two possible "back end" deep learning frameworks, Tensorflow (Abadi, Agarwal, Barham, Brevdo, Chen, Citro, Corrado, Davis, Dean, Devin, et al., 2016) and Theano (Bergstra, Breuleux, Bastien, Lamblin, Pascanu, Desjardins, Turian, Warde-Farley, & Bengio, 2010). The operator sequences can also be evaluated and differentiated on a backend implemented in the SciPy sparse-matrix package (Jones, Oliphant, & Peterson, 2014), which includes only the few operations actually needed for inference, and a simple gradient-descent optimizer. Henceforth this implementation will be called the *SciPy backend*.

The SciPy's main advantage is that it makes more use of sparse-matrix representations. In all the implementations, the matrices that correspond to DB relations are sparse. The messages corresponding to a one-hot variable binding, or the possible bindings to a variable, are sparse vectors in the SciPy version, but dense vectors in the Tensorflow and Theano versions, to allow use of GPU implementations of multiplication of dense vectors and sparse matrices. All implementations also support grouping examples into minibatches, in which case the message vectors become matrices with a number of rows equal to minibatch size—this is discussed further in Section 5.1.

TensorLog compiles query response functions on demand, i.e., only as needed to answer queries or train. In TensorLog the parameters Θ are partitioned by the predicate they are associated with, making it possible to learn parameters for any selected subset of database predicates, while keeping the remainder fixed. At compilation time, TensorLog also converts the unit facts that comprise the database into a set of sparse matrices of dimension $|\mathcal{C}| \times |\mathcal{C}|$. Since these are sparse, the total size of the matrices is $|\mathcal{DB}|$.

Execution. Given a query $Q = \mathtt{q(c,Y)}$, TensorLog first compiles the query into a series of abstract message-passing operators, as described above. These message-passing operators are then compiled into a *computation graph* (Bergstra et al., 2010; Abadi et al., 2016) for

the backend system, say Tensorflow. This graph has one "placeholder" node which can hold the input c, and also a designated node for the output, Y, called the *inference node*. Both input and output will be encoded as $|\mathcal{C}|$ dimensional vectors: the input is a one-hot vector $\mathbf{u}_c$, and the output is a vector $\hat{\mathbf{v}}_{Y|c}$ where the value of the d-th component indicates the score of `q(c,d)`. The computation graph can be executed by Tensorflow, by supplying a value for the input "placeholder" value and then requesting the value of the inference node. It is also possible to use a matrix as the input, where each row in the matrix is a separate input, and obtain a matrix of corresponding outputs, which allows several queries to be executed in parallel when execution is performed on a GPU.

Learning. For learning, the computation graph is extended by adding a second "placeholder" node corresponding to a desired target output $\mathbf{v}_{Y|c}$, and extending the graph to compute a (differentiable) loss function, which depends on the inference node and the target output. Defining the loss is usually only a few lines in Tensorflow, but TensorLog will automatically construct a default loss function, for unregularized cross-entropy loss after softmax normalization. More precisely, for an input $\mathbf{u}_c$ and target vector of desired probabilities $\mathbf{y}$, let $f^p_{\mathtt{io}}$ be defined in Eq 8. The default loss[5] is thus

$$\text{loss}(\mathbf{u}_c, \mathbf{y}) \equiv -\mathbf{y} \log(f^p_{\mathtt{io}}(\mathbf{u}_c)) - (1 - \mathbf{y}) \log(1 - f^p_{\mathtt{io}}(\mathbf{u}_c))$$

evaluated over the components of $f^p_{\mathtt{io}}(\mathbf{u}_c)$ that have non-zero values (i.e., which correspond to answers derivable from some proof). Since all the operations in the loss and inference steps are differentiable, the computation graph can also be differentiated automatically by Tensorflow: in particular, given a pair of inputs $(\mathbf{u}_c, \mathbf{v}_{Y|c})$ to the loss function, Tensorflow can compute the gradient of the loss with respect to any other graph node. Importantly, the gradient can be computed with respect to any of the nodes that correspond to the sparse matrixes derived from the database, allowing one to optimize the database to reduce loss.

Tensorflow and other deep-learning frameworks are highly efficient for matrix computations and gradient-descent based learning, with support for using sophisticated optimizers and efficient streaming training procedures. However, sparse matrix operations are not as well supported on Tensorflow: for instance, one cannot matrix-multiply two sparse matrices, and one cannot find the gradient of a sparse matrix directly. Another limitation is that constrained optimization is difficult: in the case of TensorLog, one would like to constrain the confidence for database facts to be non-negative.

To address these limitations, we convert the training data to dense vectors just before and after inference, and also define each sparse database matrix $\mathbf{M}_p$ to be a subgraph of the computation graph, where $\mathbf{M}_p$ is defined as a the output of the Tensorflow `SparseTensor` function, which inputs three dense arrays, holding the non-zero matrix values $\mathbf{v}_p$, and the row and column indices of non-zero positions, respectively. Each array of non-zero matrix values $\mathbf{v}_p$ is in turn defined as a "softplus" function[6] applied to another vector $\mathbf{v}'_p$. In learning, the $\mathbf{v}'_p$ vectors are updated during gradient descent. After learning, if necessary, the updated sparse matrices $\mathbf{M}_p$ can be recomputed and stored.

It should be emphasized that *the use of sparse matrices to store the $\boldsymbol{M}_p$ matrices is absolutely essential for effiency.* If dense matrices were used, then the size of the matrices

5. To avoid clutter, we present the loss for one example, but in learning, the loss optimized by the system is aggregated over all examples in the training dataset.
6. The softplus function is $f(x) = \ln(1 + \exp(x))$, applied componentwise to a vector.

```
tlog = tensorlog.simple.Compiler(db="data.db", prog="rules.tlog")
train_data = tlog.load_dataset("train.exam")
test_data = tlog.load_dataset("test.exam")
# data is stored dictionary mapping a function specification, like p_io,
# to a pair X, Y. The rows of X are possible inputs f^p_io, and the rows of
# Y are desired outputs.
function_spec = train_data.keys()[0]
# assume only one function spec
X,Y = train_data[function_spec]

# construct a tensorflow version of the loss function, and function used for inference
unregularized_loss = tlog.loss(function_spec)
f = tlog.inference(function_spec)
# add regularization terms to the loss
regularized_loss = unregularized_loss
for weight in tlog.trainable_db_variables(function_spec):
    regularized_loss = regularized_loss + tf.reduce_sum(tf.abs(weights))*0.01 # L1 penalty

# set up optimizer and inputs to the optimizer
optimizer = tf.train.AdagradOptimizer(rate)
train_step = optimizer.minimize(regularized_loss)
# inputs are a dictionary, with keys that name the appropriate variables used in the loss function
train_step_input = {}
train_step_input[tlog.input_placeholder_name(function_spec)] = X
train_step_input[tlog.target_output_placeholder_name(function_spec)] = Y

# run the optimizer for 10 epochs
session = tf.Session()
session.run(tf.global_variables_initializer())
for i in range(10):
    session.run(train_step, feed_dict=train_step_input)

# now run the learned function on some new data
result = session.run(f, feed_dict={tlog.input_placeholder_name(function_spec): X2})
```

Figure 6: Sample code for using TensorLog within Tensorflow. This code minimizes an alternative version of the loss function which includes an L1 penalty on the weights.

would be $O(|\mathcal{C}|^2)$ which is impractical for reasonable-size KBs—however, sparse matrices will store all the needed parameters in size linear in the number of facts in $\mathcal{DB}$.

Typed predicates. One practically important extension to the language for the Tensorflow and Theano targets was to include machinery for declaring types for the arguments of database predicates, and inferring these types for logic programs: for instance, for the sample program of Figure 1, one might include declarations like `actedIn(actor,film)` or `indicatesLabel(ngram,questionLabel)`. Typing reduces the size of the message vectors by a large constant factor, which increases the potential minibatch size and speeds up run-time by a similar factor.

Constraining the optimizer. TensorLog's learning changes the numeric score θ_f of every soft KG fact f using gradient descent. Under the proof-counting semantics used in TensorLog, a fact with a score of $\theta_f > 1$ could be semantically meaningful: for instance for $f =$ `costar(ginger_rogers,fred_astaire)` one might plausibly set θ_f to the number of movies those actors appeared in together. However it is not semantically meaningful to allow θ_f to be negative. To prevent this, before learning, for each KG parameter θ_f, we replace each occurrence of θ_f with $h(\tilde{\theta}_f)$ for the function $h = \ln(1 + e^x)$ (the "softplus" function), where $\tilde{\theta}_f \equiv h^{-1}(\theta_f)$. Unconstrained optimization is then performed to optimize the value of $\tilde{\theta}_f$ to some $\tilde{\theta}^*_f$. After learning, we update θ_f to be $h(\tilde{\theta}^*_f)$, which is always non-negative.

Extension to multi-objective learning. For learning, TensorLog's training data consists of a set of queries $p(c_1, Y), \ldots, p(c_m, Y)$, and a corresponding set of desired outputs $\mathbf{v}_{Y|c_1}, \ldots, \mathbf{v}_{Y|c_m}$. It is possible to train with examples of multiple predicates: for instance, with the example program of Figure 1, one could include training examples for both `answer` and `matches`.

Alternative semantics for query responses. One natural extension would address a limitation of the SLP semantics, namely, that the weighting of answers relative to a query sometimes leads to a loss of information. For example, suppose the answers to `father(joe,Y)` are two facts `father(joe,harry)` and `father(joe,fred)`, each with weight 0.5. This answer does not distinguish between a world in which `joe`'s paternity is uncertain, and a world in which `joe` has two fathers. One possible solution is to learn parameters that set an appropriate soft threshold on each element of w_Q, e.g., to redefine f^p as

$$f^p(\mathbf{u})) = \text{sigmoid}(g^p(\mathbf{u}) + b^p)$$

where $\text{sigmoid}(x) = 1/(1+e^{-x})$ and b^p is a bias term. This extension illustrates an advantage of being able to embed TensorLog inferences in a deep network, and will be discussed further in Section 5.2.

Extension to call out to the host infrastructure. A second extension is to allow TensorLog functions to "call out" to the backend language. Suppose, for example, we wish to replace the `classification` predicate in the example program of Figure 1 with a Tensorflow model, e.g., a multilayer perceptron, and that `buildMLP(q)` is function that constructs an expression which evaluates the MLP on input `q`. We can instruct the compiler to include this model in place of the usual function $g^{\texttt{classification}}_{\texttt{io}}$ as follows:

```
plugins = tensorlog.program.Plugins()
plugins.define("classification/io", buildMLP)
tlog = simple.Compiler(db="data.db", prog="rules.tlog", plugins=plugins)
```

This extension means that one can write logical rules over arbitrary neurally-defined low-level predicates, rather than merely over KB facts. We note that the compilation approach also makes it easy to export a TensorLog predicate (e.g., the `answer` predicate defined by the logic) to a deep learner, as a function which maps a question to possible answers and their confidences. This might be useful in building a still more complex non-logical model (e.g., a dialog agent which makes use of question-answering as a subroutine.)

An example. Figure 6 illustrates how to one might configure TensorLog, showing how to specify the function to optimize and data to train against. Because modern deep-learning frameworks are quite powerful, it is relatively easy to modify this sort of script to use the cross-compiled functions produced by TensorLog in a different way. As an example, in the figure, we show how to add L1-regularization to TensorLog's loss function and then train using the Tensorflow backend.

4. Related Work

Below we will survey several different lines of related prior work.

4.1 Learning Relations Defined by Combinations of Paths

There is a long tradition in AI of relational learning techniques in which a relation between x and y is defined in terms of the paths in a KB that link x and y (e.g., Richards & Mooney, 1992), where a *path* is defined as a sequence of relation names to follow: for instance one might define `uncle(x,y)` as $\pi_1 \vee \pi_2$ where $\pi_1 =$ `parent,brother` and $\pi_2 =$ `aunt,husband`. Paths have the advantage that one can often tractably enumerate all the short paths between x and y.

For example, the *path ranking algorithm* (PRA) learned relation definitions using logistic regression to combine path features (Lao, Mitchell, & Cohen, 2011). PRA was later extended to handle large sets of edge labels by clustering edge types (Gardner, Talukdar, Kisiel, & Mitchell, 2013) and by introducing "soft matching" for edge labels (Gardner, Talukdar, Krishnamurthy, & Mitchell, 2014). In all of these systems the classifier that is learned is specific to the particular relation being defined: e.g., there is would be separate classifiers for `uncle` queries with mode `io` and `uncle` queries with mode `oi`, and these would be separate from the classifiers for `aunt` queries.

Some recent extensions to PRA use neural network path-based classifiers. For instance (Neelakantan, Roth, & McCallum, 2015) learns to score pairs (τ, π), where π is a path and τ the name of a relation to learn. Let S_{xy} be the set of paths connecting x and y and $f(\tau, \pi)$ the learned scoring function. Their method estimates $P((x, y) \in \tau)$ with $\max_{\pi \in S_{xy}} f(\tau, \pi)$. This formulation allows f to exploit modern representation-learning methods—in particular, π is encoded with a recurrent neural network—and also allows parameter-sharing across different relations τ. Later work (Das, Neelakantan, Belanger, & McCallum, 2017b) improves on this approach by adding entity types to the path, where entity types are learned along with path weights. In a similar vein (Chen, Xiong, Yan, & Wang, 2018) uses a variational model to find a distribution over paths associated with a relation τ.

Superficially, TensorLog and these approaches seem related, as there is often a close connection between the chains of messages generated by unrolling belief propagation (e.g., the message chains for rules r1 and r2 in Table 1) and the machinery needed to follow paths in a KB. However, although many of our example programs are paths, not every TensorLog program defining a predicate $p(X, Y)$ can be defined only in terms of paths that lead from X to Y, as polytrees generalize paths—e.g., consider rule 3 in Figure 2, or the rules for `classification` in Figure 1.

The research focus of these systems is also quite different from the focus of this paper. TensorLog is a language for allowing human users to specify a probabilistic logic program

conveniently, in as expressive a way as possible: in contrast, the systems above are learning systems designed to learn to classify pairs of entities. TensorLog can be used to adjust weights in a KB: in contrast, the systems above cannot. TensorLog cannot be used for relational learning unless a logical theory defining the relation to be learned already exists: in contrast, for many of the systems above, it is hard to see how a classifier/theory can be produced except from training data.

A final distinction is that TensorLog can learn KB weights based on following long paths (e.g., see Table 5 for results on 22-step paths): in contrast, the systems above all require enumerating all valid paths between a pair (x, y) before classifying it, so they are limited to short paths. We note, however, that concurrently with this work, some approaches have been developed which generate paths dynamically as part of the learning process, either by reinforcement learning (Das, Dhuliawala, Zaheer, Vilnis, Durugkar, Krishnamurthy, Smola, & McCallum, 2017a; Xiong, Hoang, & Wang, 2017) or by an LSTM controller and a memory mechanism (Yang, Yang, & Cohen, 2017). These systems are perhaps more comparable to TensorLog in this particular, although as yet these systems have been experimentally demonstrated only on relatively short paths.

4.2 Hybrid Logical/Neural Systems

There is a long tradition of embedding logical expressions in neural networks for the purpose of learning, but generally this is done indirectly, by conversion of the logic to a boolean formula, rather than developing a differentiable theorem-proving mechanism, as considered here. Embedding logic into a network may lead to a useful architecture (Towell et al., 1990) or regularizer (Rocktäschel et al., 2015; Hu et al., 2016). Such embedding schemes require explicitly constructing explicit network nodes for all possible inferences and hence would be much more expensive than the inference scheme described here. Other well-known work in this area includes connectionist model generation (Bader, Hitzler, & Hölldobler, 2008), lifted relational neural networks (Sourek et al., 2015) and logic tensor networks (Serafini & d'Avila Garcez, 2016). These models all ground first-order theories to a neural network and like the explicitly grounded probabilistic first-order languages discussed below, these systems cannot efficiently handle rules containing many logical variables, such as the one in Equation 2.

Recently, researchers (Rocktäschel & Riedel, 2017) have proposed a "differentiable theorem prover", in which a proof for an example is unrolled into a network. Their system includes representation-learning as a component, as well as a template-instantiation approach like that of (Wang, Mazaitis, & Cohen, 2014), allowing structure learning. However, unlike the case for TensorLog, the proof procedure can produce very large proof trees, leading to proofs (and neural networks) that are of size exponential in the KB. The system also does not employ the sparse-matrix structures used by TensorLog, which act as indexes allowing one to efficiently perform chains of reasoning, and hence it cannot scale to KBs as large as those TensorLog can process. The largest published experiments with the system of (Rocktäschel & Riedel, 2017) use KBs of size less than 11,000, while below, we report results on KBs with tens of millions of facts and millions of entities.

In (Evans & Grefenstette, 2018) a "differentiable inductive logic" framework is proposed. Like TensorLog their approach is end-to-end differentiable, and involves assigning weights

to rules based on their performance on test data. In their case rules are generated by a set of templates. The differentiable inductive logic method produces good results on a number of traditional inductive logic programming (ILP) tasks, where one must infer a latent program structure, but is much more memory-intensive than TensorLog, including multiple parameters for each *potential* ground database fact (Evans & Grefenstette, 2018, Appendix E).

Another recent paper (Andreas, Rohrbach, Darrell, & Klein, 2016) describes a system in which non-logical but compositionally defined expressions are converted to neural components for question-answering tasks. These systems avoid computational issues associated with grounding by making the computational components much more like functions, with a small number of inputs to each and no search associated with inference.

4.3 Explicitly Grounded Probabilistic First-Order Languages

Many first-order probabilistic models are implemented by "grounding", i.e., conversion to a more traditional representation. In the context of a deductive DB, a rule can be considered as a finite disjunction over ground instances: for instance, the rule

```
p(X,Y) :- q(Y,Z),r(Z,Y)
```

is equivalent to

$$\exists x \in \mathcal{C}, y \in \mathcal{C}, x \in \mathcal{C} : \mathtt{p}(x,y) \vee \neg \mathtt{q}(y,z) \vee \neg \mathtt{r}(Z,Y)$$

For example, Markov logic networks (MLNs) are a widely-used probabilistic first-order model (Richardson & Domingos, 2006) in which a Bernoulli random variable is associated with each *potential* ground database fact (e.g., in the binary-predicate case, there would be a random variable for each possible $p(a,b)$ where a and b are any facts in the database and p is any binary predicate) and each ground instance of a clause is a factor. The Markov field built by an MLN is hence of size $O(|\mathcal{C}|^2)$ for binary predicates, which is much larger than the factor graphs used by TensorLog, which are of size linear in the size of the theory. In our experiments we compare to ProPPR, which has been elsewhere compared extensively to MLNs.

Inference on the Markov field can also be expensive, which motivated the development of probabilistic similarity logic (PSL), (Brocheler, Mihalkova, & Getoor, 2010) a MLN variant which uses a more tractible hinge loss. However, any grounded model for a first-order theory can still be very large, limiting the scalability of such techniques.

4.4 Stochastic Logic Programs and ProPPR

As noted above, TensorLog is very closely related to stochastic logic programs (SLPs) (Cussens, 2001). In an SLP, a probabilistic process is associated with a top-down theorem-prover: i.e., each clause r used in a derivation has an associated probability θ_r. Let $N(r,E)$ be the number of times r was used in deriving the explanation E: then in SLPs, $\Pr_{\mathtt{SLP}}(f) = \frac{1}{Z}\sum_{E \in \mathcal{E}x(f)} \prod_r \theta_r^{N(r,E)}$. The same probability distribution can be generated by TensorLog if (1) for each rule r, the body of r is prefixed with the literals `assign(RuleId,r),weighted(RuleId)`, where r is a unique identifier for the rule and (2) Θ is constructed so that $\theta_f = 1$ for ordinary database facts f, and $\theta_{\mathtt{weighted(r)}} = \theta'_{\mathtt{r}}$, where Θ' is the parameters for a SLP.

SLPs can be *normalized* or *unnormalized*. In normalized SLPs, Θ is defined so that for each set of clauses S_p of clauses with the same predicate symbol p in the head, $\sum_{r \in S_p} \theta_r = 1$. TensorLog can represent both normalized and unnormalized SLPs (although clearly learning must be appropriately constrained to learn parameters for normalized SLPs.) Normalized SLPs generalize probabilistic context-free grammars, and unnormalized SLPs can express Bayesian networks or Markov random fields (Cussens, 2001).

ProPPR (Wang et al., 2013) is a variant of SLPs in which (1) the stochastic proof-generation process is augmented with a reset, and (2) the transitional probabilities are based on a normalized soft-thresholded linear weighting of features. The first extension to SLPs can be easily modeled in TensorLog, but the second cannot: the equivalent of ProPPR's clause-specific features can be incorporated, but they are globally normalized, not locally normalized as in ProPPR.

ProPPR also includes an approximate grounding procedure which generates networks of bounded size. Asymptotic analysis suggests that ProPPR should be faster for very large databases and small numbers of training examples (assuming moderate values of ϵ and α are feasible to use), but that TensorLog should be faster with large numbers of training examples and moderate-sized databases. This is discussed further in the experimental section.

5. Experiments

To evaluate our proposal we conducted experiments to measure the scalability of the proposed algorithm, and to demonstrate how it can be integrated with Tensorflow. We also evaluated learners based on TensorLog on two types of benchmark tasks: relational learning tasks, and query-answering against a KB.

5.1 Efficiency and Scalability

One of the claimed contributions of this paper is identifying a subset of first-order logic which is expressive, but tractible. In particular, inference and learning in TensorLog should scale well when the number of facts and entities in the KB increases. To substantiate this claim, in this section we will conduct systematic experiments using similar inference and learning tasks on KBs of varying size. Another claimed advantage of TensorLog is that the inference algorithm we developed can be compiled to sequences of differentiable matrix operations, which are efficiently supported by deep learning frameworks. To substantiate this claim we will compare inference and learning times on the SciPy backend (which does not exploit an existing platfortm) with inference and learning times on the Tensorflow backend.

Inference time. We evaluated TensorLog's scalability on both inference and learning tasks. We considered two artificial problems which can be scaled to arbitrary size, both suggested by problems described in (Fierens et al., 2015). One is a version of the "friends and smokers" problem, a simplified model of social influence. Following (Fierens et al., 2015) graphs were artificially generated using a preferential attachment model. (The details of our data generation procedure and the theories used are described in Appendix C.) The inference times are averaged over four different predicates, with 100 sample query entities for each predicate, and we repeated the experiment for artificial graphs of various sizes, ranging from 400 nodes to 2 million. The second task from (Fierens et al., 2015) is intended to test performance on deeply recursive tasks. The goal here is to compute fixed-depth

Friends and Smokers			Grid Transitive Closure		
Name	# Entities	# Facts	Name	# Entities	# Facts
FS100	400	5,060	GR10	100	784
FS1k	4,000	48,260	GR25	625	5,329
FS10k	40,000	480,260	GR50	2,500	21,904
FS100k	400,000	4800,260	GR100	10,000	88,804
FS500K	2,000,000	24,000,260	GR200	40,000	357,604

Table 2: Graphs used in scalability experiments.

Graph	ProPPR	SciPy			Tensorflow CPU		Tensorflow GPU	
		none	b=25	b=250	b=25	b=250	b=25	b=250
FS100	1392.8	73.08	1247.64	–	202.29	–	452.53	–
FS1k	1310.6	71.40	1183.65	**3635.62**	143.34	926.22	198.99	1552.55
FS10k	*1190.8*	68.39	551.53	**907.64**	44.34	237.26	67.95	314.10
FS100k	*236.1*	33.19	93.33	**99.44**	5.32	24.81	11.06	37.72
FS500k	*178.4*	12.16	16.72	**17.10**	–	–	–	–
GR10	43.1	75.1	**567.6**	–	250.3	–	204.8	–
GR25	83.8	68.8	325.8	1264.2	174.9	1159.4	187.9	**1826.5**
GR50	108.1	47.2	99.4	466.0	67.9	466.8	85.5	**872.8**
GR100	117.3	11.3	12.3	108.6	10.1	88.2	19.3	**191.2**
GR200	*116.6*	0.9	1.0	8.5	0.89	7.65	1.6	**16.4**

Table 3: Queries per second on inference tasks, as KB and minibatch sizes vary. Times for the fastest exact inference scheme are put in bold, and if ProPPR's approximate scheme is yet faster, it is italicized. Missing results indicate problems that exceed Tensorflow's system limits, or else where the minibatch size is larger than the number of possible instances.

Graph	SciPy		Tensorflow CPU		Tensorflow GPU		GPU Speedup	
	Time	Acc	Time	Acc	Time	Acc	*vs* SciPy	*vs* CPU
GR10	11.6	0.90	1.23	**0.85**	**0.97**	0.80	12.0	1.3
GR25	2544.7	0.98	24.88	**1.00**	**4.77**	**1.00**	533.3	5.2
GR50	>10000	–	296.0	0.95	**30.7**	**0.97**	–	9.6
GR100	>10000	–	3203.6	0.98	**392.9**	**1.00**	–	8.2
GR200	>10000	–	–	–	–	–	–	–

Table 4: Accuracy on test data and learning time in seconds for 50 epochs of learning on grid navigation tasks of fixed difficulty, with a minibatch size of 100 and maximum depth of 10.

transitive closure on a grid: in (Fierens et al., 2015) a 16-by-16 grid was used, with a maximum path length of 10. We used the same maximum path length, but again varied the grid size, ranging from a 10-by-10 grid to a 200-by-200 grid, and averaged time over batches of b random queries, for $b = 25$ and $b = 250$. These KBs are summarized in Table 2.

We evaluated performance using the SciPy backend, which is based on SciPy (Jones et al., 2014) sparse matrices, and the Tensorflow backend. Because Tensorflow has limited support for sparse matrices, with the Tensorflow backend we use dense vectors to encode distributions over entities, and sparse matrices only to encode KB relations. Since the SciPy backend also uses sparse matrices for entity distributions, it is potentially more efficient for shallow proofs over large KBs, where only a small fraction of all entities need to have non-zero weights in a distribution. The SciPy backend also circumvents some system limits in Tensorflow (in particular, Tensorflow sparse matrices are limited in size.) For Tensorflow we evaluated performance with CPU only and also with GPUs (see Appendex C for details). Inference time was measured by queries per second.

We also varied the *minibatch size*, which usually greatly affects performance on matrix-oriented learners. When a minibatch size of b is used, then b queries with the same predicate can be processed as a group, by grouping the associated b entities into a single matrix, and all the message-passing associated with these entities is performed by a single sequence of matrix operations. This approach is not asymptotically faster but is often much faster in practice, especially when processing groups of simple queries. It also makes it possible to better exploit the parallelism of a GPU. Both the SciPy backend and the Tensorflow backend support minibatches. We explored minibatches of size $b = 25$ for the social influence task and minibatches of sizes $b = 25$ and $b = 250$ for the transitive closure task.[7]

The experimental results, summarized in Table 3, illustrate some of the computational advantages of using neural-network infrastructure for probabilistic first-order inference, and the Tensorflow backend in particular. For moderate-size KBs with a few tens of thousands or hundreds of thousands of facts (e.g., FS1k and FS10k) even small minibatches provide substantial speedup on shallow proofs—nearly 50-fold for FS1k—and this speedup is apparent even on conventional machines. However, on larger graphs there is little advantage to minibatch computation.

For the deep proofs associated with the grid tasks we see a similar effects. Minibatches give a substantial speedup on smaller KBs, and less of a speedup on larger ones: however, even for GR200, one can obtain nearly a factor of nine speedup with large minibatches, even on conventiobnal CPUs. As expected, exploiting sparsity in entity distributions is less useful, so there is little advantage in the SciPy backend over Tensorflow. On these tasks there is also a clear advantage to exploiting GPUs, with inference nearly twice as fast as on CPUs for the largest graphs, leading to a total speedup of more than a factor of 17. As we will see below, however, the advantages of GPU processing are much more pronounced on learning tasks, when inferences are repeated many times.

Discussion of inference time results. As noted above, ProPPR is a closely related probabilistic logic which includes an approximate theorem prover. ProPPR's theorem prover is

7. For the smallest KBs, there were not enough examples to populate a batch of 250.

Grid Size	Max Depth	# Graph Nodes		Acc		Time (30 epochs)	
		SciPy	TF	SciPy	TF	SciPy	TF
16	10	68	2696	**99.9**	97.2	37.6 sec	**1.1 sec**
18	12	80	3164	93.9	**96.9**	126.1 sec	**1.8 sec**
20	14	92	3632	25.2	**99.1**	144.9 sec	**2.8 sec**
22	16	104	4100	8.6	**98.4**	83.8 sec	**4.2 sec**
24	18	116	4568	**2.4**	0.0	611.7 sec	**6.3 sec**

Table 5: Learning for the grid navigation task with SciPy and Tensorflow (TF) backends, averaged over 10 trials for each datapoint, with minimal hyperparameter tuning.

also based on a very efficient abstract machine (Ait-Kaci, 1991). As predicted by analysis, ProPPR is faster for very large KBs.[8]

Both ProPPR and TensorLog are extremely fast relative to probabilistic logics using a possible-worlds semantics. For instance, (Fierens et al., 2015) reports inference times for ProbLog2, a mature system that uses the tuple-independence model on a smaller 20-node version of the social influence task, and transitive closure on a 16-by-16 grid. ProbLog2 requires 40-50 seconds for the social influence task and 100-120 seconds for transitive-closure tasks. Of course, TensorLog implements a much more restricted logic. However, for those tasks that *can* be solved with both systems, TensorLog is many orders of magnitude faster, and the tasks that can be modeled with TensorLog do include a number of practical problems, as shown in Sections 5.3 and 5.4 below.

Learning time. To evaluate learning performance as KB sizes increase, we derived a learning task from the transitive closure task, which we will call below the *grid navigation task.* On this task, there is one predicate, `path(X,Y)`, which is true if X and Y are connected by a path shorter than the maximum depth of recursion. Since the maximum depth is 10, for any query cell a there will be about 200 Y's for which `path(a,Y)` is proveable. We produced a dataset where for each a, there is one designated "landmark" node y_a that is preferred `path(`a, y_a`)`. (See Appendix C for details.) The learning task is to adjust the weights of the `edge` facts—the edges of graph defined by the grid—so that the score of `path(`a, y_a`)` is higher than every other proveable fact `path(`a, y'`)`. Thus, for an n-by-n grid, there are n^2 possible queries which are possible, with one correct answer for each query. We picked 1/3 of these queries as test data, and the remainder as train, and optimized training loss for 50 epochs using `AdagradOptimizer` with a default learning rate of 1.0 and a minibatch size of 100. Accuracies are comparable for all configurations of the system, and except for the small 10-by-10 grid, they are in the high 90's.

Tensorflow and other neural-network infrastructures are optimized for learning, in which the same sorts of inference tasks are repeated many times. This is reflected in speed of learning, especially when GPUs are used: on the 25-by-25 grid, learning is more than 500 times faster with GPUs than using the SciPy backend.

8. The experiments with ProPPR use the default parameters of $\epsilon = 0.0001$ and $\alpha = 0.1$ for the approximate theorem-proving.

The effect of inference complexity on learning. Although the grid navigation task seems simple, it is quite difficult for probabilistic logics, because recursive theories lead to large, deep proofs. While TensorLog's inference schemes scale well as KB size increases, unrolling deeply recursive proofs leads to very large graphs, especially after they are compiled to the relatively fine-grained operations used in Tensorflow. The computation graphs are also very deep, which potentially leads to problems in optimization; this is a potential advantage for systems that compile to existing deep learning frameworks, since they are likely to contain more sophisticated optimization tools, and also better support for manually or automatically tuning hyperparameters associated with the optimizer (Snoek, Larochelle, & Adams, 2012; Golovin, Solnik, Moitra, Kochanski, Karro, & Sculley, 2017).

To explore these issues, we formulated a different series of grid navigation tasks, which vary in difficulty as well as size. In all tasks, the goal is to navigate from anywhere on the grid to a specified designated landmark node, but the landmark node for a is the extreme corner closest to a—i.e., on an n-by-n grid, one of the corners (1,1), (1,n), (n,1) or (n,n). Hence as grid size increases, the path from a to a_y gets longer, and a larger maximum depth of recursive is needed. Table 5 shows how the size of the compiled sequence of message-passing operations grows as grid size and maximum depth increase.

As a measure of how difficult optimization becomes as the learning task is varied, we tuned the SciPy optimizer (a simple fixed-rate gradient descent method) so that it converged reliably on the smallest of the sample tasks. For the SciPy variant, we kept those parameters fixed as the task complexity changed. For the Tensorflow variant, we used for all tasks the `AdagradOptimizer` with a default learning rate of 1.0, which our personal experience has shown to be fairly effective on a wide range of problems. We used 30 epochs of optimization for both optimizers (the value chosen by tuning for the SciPy backend's optimizer).

The accuracy and learning-time results are shown in Table 5. Although they do not completely eliminate the need for hyperparameter tuning, the more sophisticated optimizers available in Tensorflow do appear to be more robust: AdaGrad performs well up to a depth of around 22, while the fixed-rate optimizer performs well only for depths 10 and 12. Once again, learning is many times faster for the Tensorflow backend, which uses a GPU.[9]

5.2 Integration with Tensorflow

The integration with Tensorflow means that TensorLog also can be relatively easily extended in many ways. In particular, it is possible to use TensorLog's weighted proof counts as inputs to a Tensorflow model. This is not merely a way to use logic as a feature for a neural system (although making this convenient is also useful), because one can also modify parameters of the logic using the same gradient descent methods normally used to learn the neural model, thus training the parameters of the probabilistic logic jointly with the neural model it is embedded in. A second type of integration is to introduce a neural model as an input to the logic. In particular, one can replace the facts that define a TensorLog relation with a numeric function to compute weights; this allows one to integrate a neural model as an input to TensorLog.

9. Learning times for the SciPy backend are quite variable for the larger sizes, because numerical instabilities often cause the optimizer to fail. In computing times we discard runs where there is overflow but not when there is underflow, which is harder to detect. The high variance accounts for the anomolously low average time for grid size 22.

Description	Baseline grid navigation
Inference	softmax($\mathbf{g}$)
Loss	crossEntropy(softmax($\mathbf{g}$), $\mathbf{y}$)
Target $\mathbf{y}$	$\mathbf{y}[f] = 1$ for single correct f and $\mathbf{y}[f] = 0$ otherwise
Edge relation	Sparse matrix $\mathbf{M}_{\text{edge}}$ of weights for facts `edge(a,b)`
Parameters learned	$\mathbf{M}_{\text{edge}}$
Accuracy	85.8%
Description	Simple non-conditional query response functions
Inference	sigma($b_1\mathbf{g} + b_2$)
Loss	crossEntropy(sigma($b_1\mathbf{g} + b_2$), $\mathbf{y}$)
Target $\mathbf{y}$	$\mathbf{y}[f] = 1$ for all correct f and $\mathbf{y}[f] = 0$ otherwise
Edge relation	Sparse matrix $\mathbf{M}_{\text{edge}}$ of weights for facts `edge(a,b)`
Parameters learned	$\mathbf{M}_{\text{edge}}, b_1, b_2$
Accuracy	98.9%
Description	Approximating tuple-independence
Inference	sigma($b_1\mathbf{g} + b_2$)
Loss	crossEntropy(sigma($b_1\mathbf{g} + b_2$), $\mathbf{y}$)
Target $\mathbf{y}$	$\mathbf{y}[f] = \Pr(f)$ under tuple-independence
Parameters learned	$\mathbf{M}_{\text{edge}}, b_1, b_2$
Initial error $\lvert\hat{p} - p\rvert$	0.576
Trained error $\lvert\hat{p} - p\rvert$	0.051
Description	Learning representations for grid cells
Inference	softmax($\mathbf{g}$)
Loss	crossEntropy(softmax($\mathbf{g}$), $\mathbf{y}$)
Target $\mathbf{y}$	$\mathbf{y}[f] = 1$ for the single correct f and $\mathbf{y}[f] = 0$ otherwise
Edge relation	softplus($\sum_{k=1}^{d} \mathbf{p}_{x_1}[k] - \mathbf{p}_{x_1}[k]) * \mathbf{M}_{\text{edge}}[x_1, x_2]$
Parameters learned	$\mathbf{p}_x$ for each grid cell x
Accuracy	97.8%

Table 6: Summary of experiments in integration of TensorLog and Tensorflow models. The vector $\mathbf{g}$ is a response vector containing unnormalized weighted proof-count values $w_Q(f)$ for every proveable answer fact f. The vector $\mathbf{y}$ is the desired response vector.

We used grid navigation as a testbed to explore several such extensions to TensorLogs. The experiments of this section were conducted on 10 by 10 grids, and are summarized in Table 6.

Integration of TensorLog with logistic regression to learn non-conditional probabilistic responses. As described above, TensorLog by default will answer a query of the form $Q = p(c, Y)$ with a appropriate conditional probability vector $\mathbf{v}_{Y|c}$ (given in Eq. 7). This is not always appriopriate—for instance, if Q is the query `uncle(bob,Y)` and `bob` has two true uncles, `harry` and `tom`, then a vector $\mathbf{v}_{Y|c}$ can at best split its weight between the two of them, which does not allow the user to distinguish between multiple entities related to `bob` by the `uncle` predicate, and uncertainty as to which of the two is `bob`'s uncle. In this sort of case we could like to assign high probability to both `tom` and `harry`.

Achieving this behavior is straightforward. Recall that above we denoted the combined weight of all proofs of Q which support fact f as $w_Q(f)$. In TensorLog, the response vector $\mathbf{v}_Q$ to the query Q is a normalized version of the weighted count of proofs for Q. However, in the implementation, a function that computes unnormalized weighted proof counts is also available, and these can be also converted to probabilities by rescaling and shifting the counts with learned parameters b_1 and b_2 and passing the result through a logistic function, rather than normalizing. Thus the revised response vector is

$$\Pr(f|Q) \equiv \text{sigma}(b_1 w_Q(f) + b_2)$$

where $\text{sigma}(x) \equiv \frac{1}{1+e^{-x}}$. One then minimizes an appropriate loss function, for example, cross-entropy of this output with respect to the desired labels. Implementing this new response semantics and weighting function requires modifying only five lines of code (see Appendix C).

In this experiment, then, we are using the *unnormalized proof counts as inputs to a logistic regression classifier* (which includes two additional scalar parameters A and B, which must now be learned). Hence this is an example of using TensorLog's proof-count machinery as an input to a more complex learning model.

To test this functionality, we generated data for a variant of the grid navigation problem, where the optimal response to a query `path(a,Y)` would give high probability to four squares in the corner closest to `a`, rather than one: e.g., if the closest corner was cell `(1,1)` then cell's `(1,1)`, `(1,2)`, `(2,1)`, and `(2,2)` would all be assigned probability close to 1, and all other cells assigned probability close to zero.

It is somewhat harder to optimize this alternative loss function. We used an ADAM optimizer (Kingma & Ba, 2014) with a learning rate of 0.1 and halted optimization when accuracy on the training set reached 100%: this took over 500 epochs on average (over 10 trials). However, learning still takes less than 10 seconds per trial, and performance after convergence is good: the average test-set accuracy for TensorLog was 98.88%. This result is summarized in the second section of Table 6. The first section describes the default use of TensorLog.

Learning non-conditional probabilistic responses that approximate tuple-independence. As another experiment in learning non-conditional loss functions, we also generated a dataset where $P($`path(x,y)`$)$ approximates the probability given by the tuple independence model of Section 2.4 with a maximum path length of 10. We accomplished this by sampling 1,000,000 interpretations (i.e., partial grids, where the each edge is present with fixed

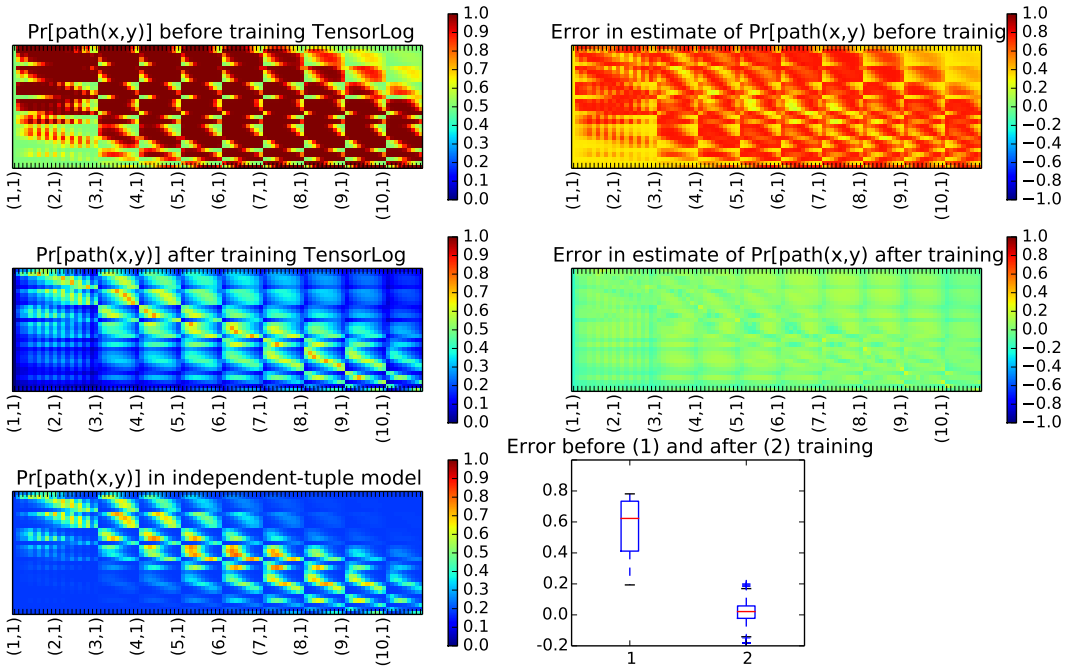

Figure 7: Learning to approximate the tuple-independence model. Top left, TensorLog's initial model for $P(\texttt{path(x,y)})$. Middle left, the model with trained edge weights. Bottom left, the target distribution, based on a sample of 1 million interpretations. Top right: the difference between TensorLog's distributions and the target distribution before training. Middle right, the difference after training. Bottom right, the distribution of errors before and after training.

probability $p = 0.25$) and recording, for each pair of cells x, y, the fraction of times that x and y are connected by a path of length 10 or less in the sample. We used the same optimization strategy to optimize for 1000 epochs, which takes about 12 seconds. Figure 7 shows that in TensorLog's semantics, the initial scores of $P(\texttt{path(x,y)}$ are quite different from those of the tuple-independence model, but that training allows it to approximate the tuple-independence model quite well.

Again, we report errors on a held-out test set of 1/3 of the possible query cells x, and train edge weights on the remaining 2/3. To evaluate we used TensorLog with the learned edge weights to compute $P(\texttt{path(x,y)})$ for each of the test query cell x and each grid cell y. These are visualized in the image plots of Figure 7: each image plot has one row for each of the 30 test query cells x, and one column for each of the 100 grid cells y, with the color encoding the value of $P(\texttt{path(x,y)})$. The mean absolute value of the difference is reduced from 0.576 to 0.051 by training, as summarized in Table 6. After training the inference times are similar to those shown in Table 3).

Applying TensorLog to predicates based on learned embeddings. To demonstrate how TensorLog can be integrated with other Tensorflow models, we constructed a final variant of the grid navigation task, where our the goal is to force the system to learn a latent representation for each cell. We built a training dataset where the desired output to every query $Q = \texttt{path(x,Y)}$ is a particular cell $y = y_{target}$, and we also limit the depth to five. The path-finding problem now becomes analogous to a reinforcement learning task, where the goal is to navigate efficiently (since depth is limited) from any query cell a to the landmark a_y. The grid was also modified so that connections "wrap around" from one edge of the grid to another,[10] so that corner and edge cells have the same pattern of connectivity as the interior cells, leaving no clues as to how to navigate but cell position.

We then replaced the $\texttt{edge}(x_1, x_2)$ facts with a computed predicate based on two learned vector representations for x_1 and x_2. Let $\mathbf{p}_x \in \mathcal{R}^d$ be the vector position for x. We would like the edge weights derived from this representation to support navigation in particular directions on the grid, e.g., toward the upper left. Notice that this cannot be done with a symmetrical weighting scheme, e.g., defining the weight of the edge from x_1 to x_2 as $\mathbf{p}_{x_1}^T \mathbf{p}_{x_1}$. We thus defined the computed edge weights $w'(x_1, x_2)$ as

$$w'(x_1, x_2) = \text{softplus}(\sum_{k=1}^{d} \mathbf{p}_{x_1}[k] - \mathbf{p}_{x_1}[k])$$

which is a simple non-negative asymmetric function. To force the weights for non-adjacent cells to be zero, we also multiply by the original sparse matrix, yielding the final formula

$$w(x_1, x_2) \equiv \text{softplus}(\sum_{k=1}^{d} \mathbf{p}_{x_1}[k] - \mathbf{p}_{x_1}[k]) * \mathbf{M}_{\text{edge}}[x_1, x_2]$$

This is integrated with TensorLog using the `plugin` construct discussed in Section 3.5. Details are given in Appendix C.

In our experiments, we used the same optimizer as in the experiments above, run for 100 epochs. We chose (1,1) as the target vertex a_y, and the embedding vectors $\mathbf{p}_x$ for each cell x

10. So, for instance, on a 10-by-10 grid, cell (4,10) would be connected to cell (4,1) as well as cell (4,9).

Task	ProPPR	TensorLog
CORA (13k facts, 10 rules)	AUC-ROC 83.2	AUC-ROC **97.6**
UMLS (5k facts, 226 rules)	Accuracy 49.8	Accuracy **52.5**
Wordnet (276k facts)		
Hypernym (46 rules)	Accuracy **93.4**	Accuracy 93.3
Hyponym (46 rules)	Accuracy 92.1	Accuracy **92.8**

Table 7: Comparison to ProPPR on small benchmark relational learning tasks.

were 2-dimensional, with each dimension initialized randomly between 1 and 10. Learning is quite accurate: averaged over 10 trials, the accuracy of this model was 97.8%, versus only 85.8% for the conventional parameterization with one independent weight per edge.[11]

Our expectation was that training would make the embeddings approximate the i, j position of a cell (since using that embedding would lead to upweighting edges leading toward the target cell.) In fact TensorLog learned a quite different embedding: both embeddings dimensions were highly correlated with each other, and each was highly correlated with distance to a_y.

5.3 Small Relational Learning Tasks

As another test, we evaluated TensorLog experimentally on several standard relational benchmark learning tasks. Here we compared TensorLog to ProPPR. We chose two traditional relational learning tasks on which ProPPR outperformed plausible competitors, such as MLNs. One was the CORA citation-matching task (from (Wang et al., 2013)), which has hand-constructed rules.[12]. A second was learning the most common relation, "affects", from UMLS, using a rule set learned by the algorithm of (Wang et al., 2014). Also, motivated by comparisons between ProPPR and embedding-based approaches to knowledge-base completion (Wang & Cohen, 2016), we compared to ProPPR on two relation-prediction tasks involving WordNet, using rules from the non-recursive theories used in (Wang & Cohen, 2016). In all of these tasks, parameters were (of course) learned on a training set separate from the test data.

The contributions of this paper include both the description of a new logic, TensorLog, and presentation of an approach to integrating TensorLog with Tensorflow. The latter contribution makes it much easier to tune parameters of a learner—e.g., selecting alternative optimizers, etc—so to isolate the value of the particular probabilistic logic used in TensorLog, we used only the SciPy backend with the default loss function and the fixed-rate stochastic gradient descent learner. We compared with the default parameters for ProPPR's learner and optimized for 30 epochs (which approximately matched the runtime for ProPPR's learning method.) Thus the only parameter remaining to set was learning rate, which was set to 0.1.

11. This may simply be due to the more efficient parameterization scheme: there are 200 parameters associated with the 2-dimensional embedding of each cell, but nearly 800 parameters distinct edges in the grid.
12. We replicated the experiments with the most recent version of ProPPR, obtaining a result slightly higher than the 2013 version's published AUC of 80.0

Original KB		Extended KB		Num Examples		
Num Tuples	Num Relations	Num Tuples	Num Relations	Train	Devel	Test
421,243	10	1,362,670	12	96,182	20,000	10,000

Table 8: Statistics concerning the WikiMovies dataset.

Method	Accuracy	Time per epoch
Subgraph/question embedding	93.5%	
Key-value memory network	93.9%	
TensorLog (1,000 training examples)	89.4%	6.1 sec
TensorLog (10,000 training examples)	94.8%	1.7 min
TensorLog (96,182 training examples)	**95.0%**	49.5 min

Table 9: Experiments with the WikiMovies dataset. The first two results are taken from (Miller et al., 2016).

Table 7 shows that the accuracy of the two systems after learning is quite comparable, even under these restrictions.

5.4 Answering Natural-Language Questions Against a KB

As a final, larger scale, experiment, we used the WikiMovies question-answering task proposed by (Miller et al., 2016). This task is similar to the one shown in Figure 1. The KB consists of over 420k tuples containing information about 10 relations and 16k movies. Some sample questions with their answers are below, with double quotes identifying KB entities.

- Question: Who acted in the movie Wise Guys?
  Answers: "Harvey Keitel", "Danny DeVito", "Joe Piscopo", ...
- Question: what is a film written by Luke Ricci?
  Answer: "How to be a Serial Killer"

We encoded the questions into the KB by extending it with two additional relations: `mentionsEntity(Q,E)`, which is true if question `Q` mentions entity `E`, and `hasFeature(Q,W)`, which is true if question `Q` contains feature `W`. The entities mentioned in a question were extracted by looking for every longest match to a name in the KB. The features of a question are simply the words in the question (minus a short stoplist).

The theory is a variant of the one given as an example in Figure 1. The main difference is that because the simple longest-exact-match heuristic described above identifies entities accurately for this dataset, we made `mentionsEntity` a hard KB predicate. We also extended the theory to handle questions with answers that are either movie-related entities (like the actors in the first example question) or movies (as in the second example). Finally, we simplified the question-classification step slightly. The final theory contains two rules and

two "soft" unary relations $\texttt{QuestionType}_{R,1}$, $\texttt{indicatesQuestionType}_{R,2}$ for each relation R in the original movie KB. For example, for the relation `directedBy` the theory has the two rules

```
answer(Question,Movie) :-
    mentionsEntity(Question,Entity), directedBy(Movie,Entity),
    hasFeature(Question,Word), indicatesQuestionType_directedBy,1(Word)
answer(Question,Entity) :-
    mentionsEntity(Question,Movie), directedBy(Movie,Entity),
    hasFeature(Question,Word), indicatesQuestionType_directedBy,2(Word)
```

The last line of each rule hence acts as a linear classifier for that rule.

For efficiency we used three distinct types of entities (question ids, entities from the original KB, and word features) and the Tensorflow backend, with minibatches of size 100 and an Adagrad optimizer with a learning rate of 0.1, running for 20 epochs, and no regularization. We compare accuracy results with two prior neural-network based methods which have been applied to this task. As shown in Table 9, TensorLog performs better than the prior state-of-the-art[13] on this task and is quite efficient.

6. Concluding Remarks

In this paper, we described a scheme to integrate probabilistic logical reasoning with the powerful infrastructure that has been developed for deep learning. The end goal is to enable deep learners to incorporate first-order probabilistic KBs, and conversely, to enable probabilistic reasoning over the outputs of deep learners. TensorLog, the system we describe here, makes this possible to do at reasonable scale using conventional neural-network infrastructure.

This paper contains several interrelated technical contributions. First, we identified a family of probabilistic deductive databases (PrDDBs) called polytree-limited stochastic deductive knowledge graphs (ptree-SDKGs) which are tractable, but still reasonably expressive. This language is a variant of SLPs, and it is maximally expressive, in that one cannot drop the polytree restriction, or switch to a possible-worlds semantics, without making inference intractible. We also argue that logics which are not tractable (i.e., are #P or worse in complexity) are unlikely to be practically incorporated into neural networks: we leave as an interesting problem for future research the question of tractable other extensions to TensorLog, e.g. to non-binary predicates.

Second, we presented an algorithm for performing inference for ptree-SDKGs, based on belief propagation. Computationally, the algorithm is quite efficient. Assuming the matrices $\mathbf{M}_p$ exist, the additional memory needed for the factor-graph G_r is linear in the size of the clause r, and hence the compilation is linear in the theory size and recursion depth. To our knowledge, use of BP for first-order inference in this manner is novel.

13. Subsequent to the original submission of the paper, a result of 96.7% accuracy was obtained (Das et al., 2017a).

Finally, we present an implementation of this logic, called TensorLog. The implementation makes it possible to both call TensorLog inference within neural models, or conversely, to call neural models within TensorLog.

The current implementation of TensorLog includes a number of restrictions. Two backends are implemented, one for Tensorflow and one for Theano, but the Tensorflow backend has been more extensively tested and evaluated. We are also exploring compilation to PyTorch[14], which supports dynamic networks. We also plan to implement support for more stable optimization (e.g., gradient clipping), and better support for debugging.

As noted above, TensorLog also makes it possible to replace components of the logic program (e.g., the `classification` or `matches` predicate) with submodels learned in the deep-learning infrastructure. Alternatively, one can export a `answer` predicate defined by the logic to a deep learner, as a function which maps a question to possible answers and their confidences; this might be useful in building a still more complex model non-logical model (e.g., a dialog agent which makes use of question-answering as a subroutine.) In this paper we explored small illustrative experiments to test these capabilities, but in future work we hope to explore them futher.

We also note that although the experiments in this paper assume that theories are given, the problem of learning programs in TensorLog is also of great interest. Some early results from the authors on this problem are discussed elsewhere (Yang et al., 2017).

Acknowledgments

Thanks to William Wang for providing some of the datasets used here; and to William Wang and many other colleagues contributed with technical discussions and advice. The author is greatful to Google for financial support, and also to NSF for their support of his work via grants CCF-1414030 and IIS-1250956.

Appendix A. Proofs

Theorem 1 *Computing $P(f|Q)$ (relative to a SDKG $\mathcal{T}, \mathcal{DB}$) for all possible answers f of the query Q is #P-hard, even if there are only two such answers, the theory contains only two non-recursive[15] clauses, and the KG contains only 13 facts.*

We will reduce counting proofs for 2PSAT to computing probabilities for SDKGs. 2PSAT is a #P-hard task where the goal is to count the number of satisfying assignments to a CNF formula with only two literals per clause, all of which are positive (Suciu et al., 2011). Hence a 2PSAT formula is of the form

$$(x_{a_1} \vee x_{b_1}) \wedge \ldots \wedge (x_{a_n} \vee \ell_{b_n})$$

where the variables are all binary variables x_i from $X = \{x_1, \ldots, x_n\}$, and each a_i and b_i is a index between 1 and n. The subformula $(x_{a_i} \vee x_{b_i})$ is called the i-th clause below.

14. pytorch.org
15. A theory is *recursive* if some proof of a query $q(x_1, y_1)$ involves a subgoal of the form $q(x_2, y_2)$, where the x's and y's are either constants or variables.

The database contains the two facts `assign_yes(yes)` and `assign_no(no)` with weight 1, and two facts `binary(0)` and `binary(1)` with weights 0.5. It also contains a definition of the predicate `either_of`, containing the following three weight 1 facts: `either_of(0,1)`, `either_of(1,0)`, and `either_of(1,1)`. There are a total of 7 facts in the database.

```
sat(Y) :-
      assign_yes(Y), binary(X_1), ..., binary(X_n),
      either_of(X_{a_1},X_{b_1}),
      ...,
      either_of(X_{a_n},X_{b_n}),
sat(Y) :-
      assign_no(Y).
```

In the first line of the first rule, an assignment to the x_i's is selected, with uniform probability. It is easy to see that the literal `either_of(X`$_{a_i}$`,X`$_{b_i}$`)` will succeed iff the i-th clause is made true by this assignment. Hence the first rule of the theory will succeed exactly k times, where k is the number of satisfying assignments for the formula. The second clause succeeds once, so

$$p = \Pr(\texttt{sat(yes)}|\texttt{sat(Y)}) = \frac{k}{k+1}$$

If p could be computed efficiently, one could solve the equation above for k and use the result to determine the number of satisfying assignments to the 2PSAT formula.

Theorem 3 *Computing $P(f)$ in the tuple-independent possible-worlds semantics for a single ground fact f is #P-hard.*

We again reduce counting assignments for 2NSAT to computation of $p = \Pr(\texttt{sat(yes)})$. In this case the DB contains n facts of the form `x`$_1$`(1)`, `x`$_2$`(1)`, ..., `x`$_n$`(1)`, all with weights 0.5, and the additional fact `assign_yes(yes)`.

We can now encode the 2PSAT formula with the following theory. For each clause i let $j1$ and $j2$ be the indices of the two literals in that clause. We construct two theory rules for each clause i:

```
sat_i(Y) :- x_{j1}(Y).
sat_i(Y) :- x_{j2}(Y).
```

Finally we add a binary tree of $O(\log(n))$ rules, each of which test success of two other subpredicates, and the last of which tests succeeds only of all the `clause`$_i$ predicates succeed. For instance, for $n = 8$, we would define `sat(Y)` as

```
sat_{1:2}(Y) :- clause_1(Y),clause_2(Y).
sat_{2:3}(Y) :- clause_2(Y),clause_3(Y).
sat_{1:4}(Y) :- sat_{1:2}(Y),sat_{2:3}(Y).
sat_{5:6}(Y) :- clause_5(Y),clause_6(Y).
sat_{7:8}(Y) :- clause_7(Y),clause_8(Y).
```

```
sat_{5:8}(Y) :- sat_{5:6}(Y),sat_{7:8}(Y).
sat(Y) :- sat_{1:4}(Y),sat_{5:8}(Y),assign_yes(Y).
```

Note that for each variable x_j, an I drawn from the database distribution may contain either `x_j(1)` or not, so there are 2^n possible interpretations. Each of these corresponds to a boolean assignment, where $x_j = 1$ in the assignment exactly when `x_j(1)` is in the corresponding interpretation. Clearly the `clause_i` predicate succeeds exactly when the i-th clause is satisfied, and hence $p = \Pr(\texttt{sat(yes)}|\mathcal{DB}, |T)$ is thus exactly $\frac{k}{2^n}$, where k is the number of satisfying assignments.

This theory is quite simple: it contains no binary predicates, and (if one tests success of all clause predicates in a tree) the rules are all very short. It is thus difficult to identify syntactic restrictions which might make proof-counting tractible for the possible-worlds scenario.

Appendix B. Discussion: SDKGs and Extensional DDBs

For SDKGs, a final connection with other logics can be made by considering a logic program that has been grounded by conversion to a boolean formulae. One simple approach to implementing a "soft" extension of a boolean logic is to evaluate the truth or falsity of a formula bottom-up, deriving a numeric confidence c for each subexpression from the confidences associated with its subparts. For instance, one might use the rules

$$\begin{aligned} c(x \wedge y) &\equiv \min(c(x), c(y)) \\ c(x \vee y) &\equiv \max(c(x), c(y)) \\ c(\neg x) &\equiv 1 - c(x) \end{aligned}$$

This approach to implementing a soft logic is is sometimes called an *extensional* approach (Suciu et al., 2011), and it is common in practical systems: PSL (Brocheler et al., 2010) uses an extensional approach, as do several recent neural approaches (Serafini & d'Avila Garcez, 2016; Hu et al., 2016).

Now consider modifying a top-down prover to produce a particular boolean formula, in which each path $v_0 \to \ldots \to v_n$ is associated with a conjunction $f_1 \wedge \ldots \wedge f_m$ of all unit-clause facts used along this path, and each answer f is associated with the disjunction of these conjunctions. Then let us compute the unnormalized weight $w_Q(f)$ using the rules

$$\begin{aligned} c(x \wedge y) &\equiv c(x) \cdot c(y) \\ c(x \vee y) &\equiv c(x) + c(y) \end{aligned}$$

(which are sufficient since no negation occurs in the formula). This (followed by normalization) can be shown to be equivalent to the SLP semantics.

Appendix C. Experimental Details

Graph generation. To generate the graphs used for the "friends and smokers" experiments in Section 5.1, we generated four subgraphs, each with n nodes, using the Barabasi-Albert

Friends and Smokers

```
stressed(P,yes) :- person(P).
influences(P1,P2) :- friends(P1,P2).
cancer_spont(P,yes) :- person(P).
cancer_smoke(P,yes) :- person(P).
smokes(X,Status) :- stress(X,Status).
smokes(X,yes) :- smokes(Y), influences(Y,X).
cancer(P,Status) :- cancer_spont(P,Status).
cancer(P,Status) :- smokes(P,Status), cancer_smoke(P,Status ).
```

Transitive Closure

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Figure 8: Theories used in the experiments.

model (Barabási & Albert, 1999). For each pair of distinct subgraphs, we then chose 25 random node pairs and connected them with edges, creating a fully-connected graph with strong subcommunities. Each subgraph had different joint distribution of smoking and cancer.

Theories. The theories used for the "friends and smokers" and transitive closure tasks are shown in Figure 8.

Hardware. For Tensorflow we evaluated performance in the CPU-only configuration on a machine with 4 16-core 2.3GHz AMD Opteron 6376 processors and 512Gb of RAM. The GPU configuration was evaluated on a machine with 2 8-core 2.4GHz Intel Xeon E5-2630 processors and a GeForce GTX TITAN X graphics card with 12 GB memory.

Grid navigation tasks used for scalability. For any cell a, let i, j be the row and column position of a in the grid. We define a_y to be cell i', j' where $i' = 10 * \lfloor i/10 \rfloor + 5$ and $j' = 10 * \lfloor j/10 \rfloor + 5$: for example for $a = (i, j) = (12, 29)$, $a_y = (15, 25)$. This means that a_y is always within distance 10 of a, and the learning task is identically repeated for each 10-by-10 subgrid of the graph, making it neither easier nor harder to learn as the graph size changes.

Facts are present that assert an edge between a and every a' such that $|i' - i| \leq 1$ and $|j' - j| \leq 1$. The initial confidence for each such fact was set to 0.2 in the learning tasks.

Non-conditional loss functions. In learning with the Tensorflow backend, one must write Python code to define the loss function and optimize it, as shown in Figure 6. To modify the loss function we replaced the lines

```
predictions = tlog.inference('path/io')
loss = tlog.loss('uncle/io')
```

with

```
logits = B1 * tlog.proofcount('path/io') + B2
```

```
predictions = tf.sigmoid(logits)
loss = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(
            logits=logits, labels=tlog.target_output_placeholder('path/io')))
```

The use of "sigmoid cross entropy" loss over the inputs of logistic function (`sigmoid`) above is a common pattern in Tensorflow for learning multiclass classifiers.

Accuracy for non-conditional responses was also measured differently. In the conditional case, we evaluate accuracy as follows: a prediction of y for `path(a,`y`)` is treated like a single example of a multiclass classification task, where the classes are the potential grid cells y: in particular, the example associated with `a` is considered correct if the correct y_a is scored highest. The accuracy of a random classifier is thus very low—for an n-by-n grid, it would be $\frac{1}{n^2}$. For the non-conditional case, we threshold the scores for `path(a,`y`)` for all y's, and consider the ones with scores higher than 0.5 as positive predictions and the others as negative predictions. We then consider each possible x, y pair as an example, which is correct if y is a desired answer for x. The accuracy of a random classifier is quite high for this case—for the 10-by-10 grid, it is 96%.

Learning embeddings with plugins. In the experiments using plugins to define computed edge functions, we used two different vectors `E1` and `E2` to encode the two dimensions of the embedding of each cell. `pathMatrix` denotes the $\mathbf{M}_{\text{edge}}$ matrix, and n denotes the number of database constants. The computed edge relation was defined as follows.

```
def embedded_edge_fun(v):
    A1 = tf.reshape(tf.tile(E1,[1,n]), [n,n])
    A2 = tf.reshape(tf.tile(E2,[1,n]), [n,n])
    Distance0 = tf.nn.softplus((tf.transpose(A1)-A1) + (tf.transpose(A2)-A2))
    return tf.matmul(v,tf.multiply(Distance0, pathMatrix))
tlog.prog.plugins.define('embedded_edge/io',embedded_edge_fun)
```

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Ait-Kaci, H. (1991). *Warren's abstract machine: a tutorial reconstruction*. MIT press.

Andreas, J., Rohrbach, M., Darrell, T., & Klein, D. (2016). Learning to compose neural networks for question answering. *CoRR*, *abs/1601.01705*.

Bader, S., Hitzler, P., & Hölldobler, S. (2008). Connectionist model generation: A first-order approach. *Neurocomputing*, *71*(13-15), 2420–2432.

Barabási, A.-L., & Albert, R. (1999). Emergence of scaling in random networks. *Science*, *286*(5439), 509–512.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., & Bengio, Y. (2010). Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf*, pp. 1–7.

Brocheler, M., Mihalkova, L., & Getoor, L. (2010). Probabilistic similarity logic. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

Chen, W., Xiong, W., Yan, X., & Wang, W. (2018). Variational knowledge graph reasoning. *arXiv preprint arXiv:1803.06581*.

Cussens, J. (2001). Parameter estimation in stochastic logic programs. *Machine Learning*, *44*(3), 245–271.

Das, R., Dhuliawala, S., Zaheer, M., Vilnis, L., Durugkar, I., Krishnamurthy, A., Smola, A., & McCallum, A. (2017a). Go for a walk and arrive at the answer: Reasoning over paths in knowledge bases using reinforcement learning. *arXiv preprint arXiv:1711.05851*.

Das, R., Neelakantan, A., Belanger, D., & McCallum, A. (2017b). Chains of reasoning over entities, relations, and text using recurrent neural networks. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, Vol. 1, pp. 132–141.

De Raedt, L., & Kersting, K. (2008). *Probabilistic inductive logic programming*. Springer.

Evans, R., & Grefenstette, E. (2018). Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, *61*, 1–64.

Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., & De Raedt, L. (2015). Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, *15*(3), 358–401.

Gardner, M., Talukdar, P., Krishnamurthy, J., & Mitchell, T. (2014). Incorporating vector space similarity in random walk inference over knowledge bases. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 397–406.

Gardner, M., Talukdar, P. P., Kisiel, B., & Mitchell, T. (2013). Improving learning and inference in a large knowledge-base using latent syntactic cues. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 833–838.

Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., & Sculley, D. (2017). Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pp. 1487–1495, New York, NY, USA. ACM.

Goodman, N., Mansinghka, V., Roy, D. M., Bonawitz, K., & Tenenbaum, J. B. (2012). Church: a language for generative models. *arXiv preprint arXiv:1206.3255*.

Gormley, M. R., Dredze, M., & Eisner, J. (2015). Approximation-aware dependency parsing by belief propagation. *Transactions of the Association for Computational Linguistics (TACL)*.

Hu, Z., Ma, X., Liu, Z., Hovy, E., & Xing, E. (2016). Harnessing deep neural networks with logic rules. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1, pp. 2410–2420.

Jones, E., Oliphant, T., & Peterson, P. (2014). *SciPy: open source scientific tools for Python*.

Kimmig, A., Mihalkova, L., & Getoor, L. (2015). Lifted graphical models: a survey. *Machine Learning*, *99*(1), 1–45.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kschischang, F. R., Frey, B. J., & Loeliger, H.-A. (2001). Factor graphs and the sum-product algorithm. *Information Theory, IEEE Transactions on*, *47*(2), 498–519.

Lao, N., Mitchell, T. M., & Cohen, W. W. (2011). Random walk inference and learning in a large scale knowledge base. In *EMNLP*, pp. 529–539. ACL.

Miller, A., Fisch, A., Dodge, J., Karimi, A.-H., Bordes, A., & Weston, J. (2016). Key-value memory networks for directly reading documents. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 1400–1409, Austin, Texas. Association for Computational Linguistics.

Muggleton, S., et al. (1996). Stochastic logic programs. *Advances in inductive logic programming*, *32*, 254–264.

Neelakantan, A., Roth, B., & McCallum, A. (2015). Compositional vector space models for knowledge base inference. In *2015 AAAI spring symposium series*.

Pearlmutter, B. A., & Siskind, J. M. (2008). Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *30*(2), 7.

Poole, D. (1997). The independent choice logic for modelling multiple agents under uncertainty. *Artificial intelligence*, *94*(1), 7–56.

Ramakrishnan, R., & Ullman, J. D. (1995). A survey of deductive database systems. *The Journal of Logic Programming*, *23*(2), 125–149.

Richards, B. L., & Mooney, R. J. (1992). Learning relations by pathfinding. In *Proceedings of the tenth national conference on Artificial intelligence*, pp. 50–55. AAAI Press.

Richardson, M., & Domingos, P. (2006). Markov logic networks. *Mach. Learn.*, *62*(1-2), 107–136.

Rocktäschel, T., & Riedel, S. (2017). End-to-end differentiable proving. In *Advances in Neural Information Processing Systems*, pp. 3791–3803.

Rocktäschel, T., Singh, S., & Riedel, S. (2015). Injecting logical background knowledge into embeddings for relation extraction. In *Proc. of ACL/HLT*.

Serafini, L., & d'Avila Garcez, A. S. (2016). Logic tensor networks: Deep learning and logical reasoning from data and knowledge. In *Proceedings of the 11th International Workshop on Neural-Symbolic Learning and Reasoning (NeSy'16) co-located with the Joint Multi-Conference on Human-Level Artificial Intelligence (HLAI 2016), New York City, NY, USA, July 16-17, 2016.*

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pp. 2951–2959.

Sourek, G., Aschenbrenner, V., Zelezný, F., & Kuzelka, O. (2015). Lifted relational neural networks. *CoRR*, *abs/1508.05128*.

Suciu, D., Olteanu, D., Ré, C., & Koch, C. (2011). Probabilistic databases. *Synthesis Lectures on Data Management*, *3*(2), 1–180.

Towell, G., Shavlik, J., & Noordewier, M. (1990). Refinement of approximate domain theories by knowledge-based artificial neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, Massachusetts. MIT Press.

Wang, W. Y., & Cohen, W. W. (2016). Learning first-order logic embeddings via matrix factorization. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, New York, NY. AAAI.

Wang, W. Y., Mazaitis, K., & Cohen, W. W. (2013). Programming with personalized PageRank: a locally groundable first-order probabilistic logic. In *Proceedings of the 22nd ACM International Conference on Conference on Information & Knowledge Management*, pp. 2129–2138. ACM.

Wang, W. Y., Mazaitis, K., & Cohen, W. W. (2014). Structure learning via parameter learning. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pp. 1199–1208. ACM.

Xiong, W., Hoang, T., & Wang, W. Y. (2017). Deeppath: A reinforcement learning method for knowledge graph reasoning. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 564–573.

Yang, F., Yang, Z., & Cohen, W. W. (2017). Differentiable learning of logical rules for knowledge base reasoning. In *Advances in Neural Information Processing Systems*, pp. 2319–2328.