TensorLog: A Differentiable Deductive Database

William W. Cohen
Department of Machine Learning
Carnegie Mellon University
Pittsburgh, PA 15213
`wcohen@cs.cmu.edu`

Abstract

Large knowledge bases (KBs) are useful in many tasks, but it is unclear how to integrate this sort of knowledge into "deep" gradient-based learning systems. To address this problem, we describe a probabilistic deductive database, called TensorLog, in which reasoning uses a differentiable process. In TensorLog, each clause in a logical theory is first converted into certain type of factor graph. Then, for each type of query to the factor graph, the message-passing steps required to perform belief propagation (BP) are "unrolled" into a function, which is differentiable. We show that these functions can be composed recursively to perform inference in non-trivial logical theories containing multiple interrelated clauses and predicates. Both compilation and inference in TensorLog are efficient: compilation is linear in theory size and proof depth, and inference is linear in database size and the number of message-passing steps used in BP. We also present experimental results with TensorLog and discuss its relationship to other first-order probabilistic logics.

1 Introduction

Large knowledge bases (KBs) have proven useful in many tasks, but it is unclear how to integrate this sort of knowledge into "deep" gradient-based learning systems. Motivated by this, we describe a probabilistic deductive database (PrDDB) system in which reasoning is performed by a differentiable process. In addition to enabling novel gradient-based learning algorithms for PrDDBs, this approach could potentially enable tight integration of logical reasoning into deep learners (or conversely, of deep learning into reasoning systems.

In a traditional deductive database (DDB), a database $\mathcal{DB}$ with a theory $\mathcal{T}$ together define a set of facts $f_1, \ldots, f_n$ which can be derived by accessing the database and reasoning using $\mathcal{T}$. As an example, Figure 1 contains a small theory and an associated database. End users can test to see if a fact f is derivable, or retrieve all derivable facts that match some query: e.g., one could test if $f = \texttt{uncle(joe,bob)}$ is derivable in the sample database, or find all values of Y such that `uncle(joe,Y)` holds. A probabilistic DDB is a "soft" extension of a DDB, where derived facts f have a numeric confidence, typically based on augmenting $\mathcal{DB}$ with a set of parameters Θ. In many existing PrDDB models computation of confidences is computationally expensive, and often not be conducive to learning the parametersb Θ.

Here we describe a probabilistic deductive database called TensorLog in which reasoning uses a differentiable process. In TensorLog, each clause in a logical theory is first converted into certain type of factor graph, in which each logical variable appearing in the clause is associated with a random variable in the factor graph, and each literal is associated with a factor (as shown in Figure 2). Then, for each type of query to the factor graph, the message-passing steps required to perform BP are "unrolled" into a function, which is differentiable. Each function will answer queries for a particular combination of evidence variables and query variables in the factor graph, which in turn corresponds to logical queries in a particular syntactic form. We also show how these functions can be composed

29th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

1. `uncle(X,Y):-child(X,W),brother(W,Y).`	`child(liam,eve)`,0.99	`infant(liam)`,0.7
2. `uncle(X,Y):-aunt(X,W),husband(W,Y).`	`child(dave,eve)`,0.99	`infant(dave)`,0.1
3. `status(X,tired):-child(W,X),infant(W).`	`child(liam,bob)`,0.75	`aunt(joe,eve)`,0.9
	`husband(eve,bob)`,0.9	`brother(eve,chip)`,0.9

Figure 1: An example database and theory. 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.

recursively to perform inference in non-trivial logical theories containing multiple interrelated clauses and predicates.

In TensorLog, compilation is linear in theory size and proof depth, and inference is linear in database size and the number of message-passing steps used in BP. Most importantly, *inference is also differentiable*, enabling gradient-based parameter learning. Formally, we can show that TensorLog subsumes some prior probabilistic logic programming models, including several variants of stochastic logic programs (SLPs) [3, 17], and approximates others [9, 5].

Below, we first present background material, then introduce our main results for differentiable inference, We then discuss related work, in particular the relationship between TensorLog and existing probabilistic logics, present experimental results, and conclude.

2 Background: Deductive and Probabilistic DBs

To begin, we review the definition for an ordinary DDB, an example of which is in Figure 1. A *database*, $\mathcal{DB}$, is a set $\{f_1, \ldots, f_N\}$ of ground facts. We focus here on DB relations which are unary or binary (e.g., from a "knowledge graph"), hence, facts will be written as $p(a, b)$ or $q(c)$ where p and q are *predicate symbols*, and a, b, c are constants from a fixed domain $\mathcal{C}$. A theory, $\mathcal{T}$, is a set of function-free Horn clauses. Clauses are written $A\text{:-}B_1, \ldots, B_k$, 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 $q(X)$, $p(X, Y)$, $p(c, Y)$, or $p(X, c)$, where X and Y are logical variables, and c is a database constant.

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 1, 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 $\mathcal{T}$. In the usual DDB semantics, a ground fact f is considered "true" iff $f \in$ *Model*$(\mathcal{DB}, T)$.

To introduce "soft" computation into this model, we add a parameter vector Θ which associates each fact $f \in \mathcal{DB}$ with a positive scalar θ_f (as shown in the example). The semantics of this parameter vary in different PrDDB models, but Θ will always define a distribution $\Pr(f|\mathcal{T}, \mathcal{DB}, \Theta)$ over the facts in *Model*$(\mathcal{T}, \mathcal{DB})$.

3 Differentiable soft reasoning

Numeric encoding of PrDDB's and queries. We will implement reasoning in PrDDB's by defining a series of numeric functions, each of finds answers to a particular family of 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 $\mathbf{v}_q$. Note that $\mathbf{M}_p$ encodes information not only about the database facts in predicate p, but also about their parameter values.

PrDDB's are commonly used test to see if a fact f is derivable, or retrieve all derivable facts that match some query: e.g., one could test if $f =$ `uncle(joe,bob)` is derivable in the sample database, or

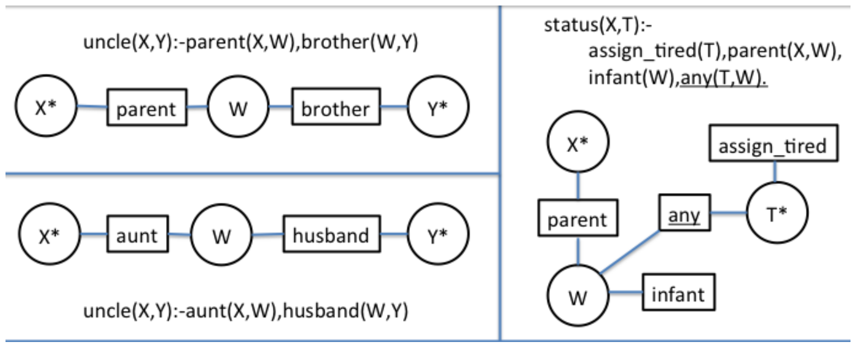

Figure 2: Examples of factor graphs for the example theory.

find all values of Y such that `uncle(joe,Y)` holds. We focus here on the latter type of query, which we call an argument-retrieval query. An *argument-retrieval query* Q is 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 `io`.

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 $d \in \mathcal{C}$, $\mathbf{v}_Y[d] = \Pr(p(c, d)|\mathcal{T}, \mathcal{DB}, \Theta)$. Alternatively (since often we care only about the relative scores of the possible answers), the system might instead return a conditional probability vector $\mathbf{v}_{Y|c}$: if $U_{p(c,Y)}$ is the set of facts f that "match" $p(c, Y)$, then $\mathbf{v}_{Y|c}[d] = \Pr(f = p(c, d)|f \in U_{p(c,Y)}, \mathcal{T}, \mathcal{DB}, \Theta)$.

Since the ultimate 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 $f^p_{\mathtt{io}}$ denotes a *query response function* for all queries with predicate p and mode `io`, i.e., queries of the form $p(c, Y)$, when given a one-hot encoding of c, $f^p_{\mathtt{io}}$ returns the appropriate conditional probability vector:

$$f^p_{\mathtt{io}}(\mathbf{u}_c) \equiv \mathbf{v}_{Y|X} \text{ where } \forall d \in C : \mathbf{v}_{Y|c}[d] = \Pr(f = p(c, d)|f \in U_{p(c,Y)}, \mathcal{T}, \mathcal{DB}, \Theta) \quad (1)$$

and similarly for $f^p_{\mathtt{oi}}$.

Syntactic restrictions. Algorithmically it will be convenient to constrain the use of constants in clauses. We introduce a special DB predicate `assign`, which will be used only in literals of the form `assign`(W,c), which in turn will be treated as literals for a special unary predicate `assign_`c. Without loss of generality, we can now assume that constants only appear in `assign` literals. For instance, the clause 3 of Figure 1 would be rewritten as

```
status(X,T):-assign_tired(T),child(X,W),infant(W).          (2)
```

We will also 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 $\mathbf{M}_{\mathtt{any}}$ need not be explicitly stored. We also constrain clause heads to contain distinct variables which all appear also in the body.

A factor graph for a one-clause program. We will start by considering a highly restricted class of theories $\mathcal{T}$, namely programs containing only one non-recursive 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. Figure 2 gives examples. The variables appearing in the clause's head are starred.

We now argue that G_r imposes a valid distribution $\Pr(f|\mathcal{T}, \mathcal{DB}, \Theta)$ over facts in *Model*$(\mathcal{T}, \mathcal{DB})$. In G_r the variables are multinomials over $\mathcal{C}$, the factors represent predicates and the graph G_r represents a distribution of *possible bindings to the logical variables in the clause* f, i.e., to possible substitutions σ. Let $W_1, \ldots, W_m$ be the variables of G_r, and for each factor/edge e let $p_e(W_{i_e}, W_{j_e})$ be the literal

```
define compileMessage(L → X):
  assume wolg that L = q(X) or L = p(X_i, X_o)
  generate a new variable name v_{L,X}
  if L = q(X) then
    emitOperation( v_{L,X} = v_q )
  else if X is the output variable X_o of L then
    v_i = compileMessage(X_i → L)
    emitOperation( v_{L,X} = v_i · M_p )
  else if X is the input variable X_i of L then
    v_o = compileMessage(X_i → L)
    emitOperation( v_{L,X} = v_o · M_p^T )
  return v_{L,X}

define compileMessage(X → L):
  if X is the input variable X then
    return u_c, the input
  else
    generate a new variable name v_X
    assume L_1, L_2, ..., L_k are the
      neighbors of X excluding L
    for i = 1, ..., k do
      v_i = compileMessage(L_i → X)
    emitOperation(v_X = v_1 ∘ ··· ∘ v_k)
    return v_X
```

Figure 3: 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.

associated with it. In the distribution defined by G_r

$$\Pr_{G_r}(W_1 = c_1, \ldots, W_m = c_m) = \frac{1}{Z} \prod_{(c_i,c_j) \in \textit{edges } e} \phi_e(c_i, c_j) = \prod_{(c_i,c_j) \in \textit{edges } e} \theta_{p_e(c_{i_e}, c_{j_e})}$$

Recall $\forall f, \theta_f > 0$, so if $\Pr(W_1 = c_1, \ldots, W_m = c_m) > 0$ then for each edge $p_e(c_{i_e}, c_{j_e}) \in \mathcal{DB}$, and hence the substitution $\sigma = \{W_1 = c_1, \ldots, W_m = c_m\}$ makes the body of clause r true. The converse is also clearly true: so G_r defines a distribution over exactly those substitutions σ that make the the body of r true.

BP over G_r can now be used to compute the conditional vectors $f^p_{\mathtt{io}}(\mathbf{u}_c)$ and $f^p_{\mathtt{oi}}(\mathbf{u}_c)$. For example to compute $f^p_{\mathtt{io}}(\mathbf{u}_c)$ 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.

However, we would like to do more: we would like to compute an explicit, differentiable, query response function, which computes $f^p_{\mathtt{io}}(\mathbf{u}_c)$. To do this we "unroll" the message-passing steps into a series of operations, following [6].

For completeness, we include in Figure 3 a sketch of the algorithm used in the current implementation of TensorLog, which makes the (strong) assumption that G_r is a tree. In the code, we found it convenient to extend the notion of input-output modes for a query, a variable X appearing in a literal $L = p(X, Y)$ in a clause body is an *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 M_p rather than M_p^Tw (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: e.g., for the sample clauses the functions returned are:

r1 $g^{r1}_{\mathtt{io}}(\vec{u}_c) = \{\ \mathbf{v}_{1,W} = \mathbf{u}_c \mathbf{M}_{\mathtt{parent}};\ \mathbf{v}_W = \mathbf{v}_{1,W};\ \mathbf{v}_{2,Y} = \mathbf{v}_W \mathbf{M}_{\mathtt{brother}};\ \mathbf{v}_Y = \mathbf{v}_{2,Y};\ \ \textbf{return } \mathbf{v}_Y \}$

r2 $g^{r2}_{\mathtt{io}}(\vec{u}_c) = \{\ \mathbf{v}_{1,W} = \mathbf{u}_c \mathbf{M}_{\mathtt{aunt}};\ \mathbf{v}_W = \mathbf{v}_{1,W};\ \mathbf{v}_{2,Y} = \mathbf{v}_W \mathbf{M}_{\mathtt{husband}};\ \mathbf{v}_Y = \mathbf{v}_{2,Y};\ \ \textbf{return } \mathbf{v}_Y \}$

r3 $g^{r3}_{\mathtt{io}}(\vec{u}_c) = \{\ \mathbf{v}_{2,W} = \mathbf{u}_c \mathbf{M}_{\mathtt{parent}};\ \mathbf{v}_{3,W} = \mathbf{v}_{\mathtt{infant}};\ \mathbf{W} = \mathbf{v}_{2,W} \circ \mathbf{v}_{3,W};$
$\mathbf{v}_{1,T} = \mathbf{v}_{\mathtt{assign\_tired}};\ \mathbf{v}_{4,T} = \mathbf{v}_W \mathbf{M}_{\mathtt{any}};\ \mathbf{T} = \mathbf{v}_{1,T} \circ \mathbf{v}_{4,T};\ \ \textbf{return } \mathbf{v}_T \}$

Here we use $g^r_{\mathtt{io}}(\vec{u}_c)$ for the unnormalized version of the query response function build from G_r, i.e.,

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

where r is the one-clause theory defining p.

Sets of factor graphs 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 cduncle, defined by rules r_1 and r_2, we can define

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

and then re-normalize. This is equivalent to building a new factor graph G, which would be approximately $\cup_i G_{ri}$, together global input and output variables, and a factor that constrains the input variables of the G_{ri}'s to be equal, and 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 $\mathbf{v}_Y = \mathbf{v}_X \mathbf{M}_q^T$ 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 some sort of 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 $g^p_{\texttt{io},d}$ to q always call the next-deeper version of the function for q, e.g., $g^q_{\texttt{io},d+1}$.

Uncertain inference rules. Notice that Θ associates confidences with *facts* in the databases, not with *clauses* in the theory. To attach a probability to a clause, a standard trick is to introduce a special clause-specific fact, and add it to the clause body **(author?)** [9]. For example, a soft version of clause 3 could be re-written as

```
status(X,tired):-assign(RuleId,c3),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 $\mathcal{T}$. TensorLog supports some special syntax to make it easy to build rules with associated weights: for instance, `status(X,tired)` :- `assign(C3,c3), weighted(C3), child(W,X), infant(W)` can be written simply as `status(X,tired)` :- `child(W,X), infant(W) {c3}`.

Discussion. Collectively, the computation performed by TensorLog's functions are equivalent to computing a set of marginals over a particular factor graph G: specifically G would be formed by using the construction for multiple clauses with the same head (described above), and then splicing in the factor graphs of subpredicates. The unnormalized messages over this graph, and their functional equivalent, can be viewed implementing a first-order version of *weighted model counting*, a well-studied problem in satisfiability.

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 theories where every 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 due to sparsity.

The current implementation of TensorLog includes many restrictions that could be relaxed: e.g., predicates must be unary or binary, only queries of the types discussed here are allowed, and every factor graph G_r must be a tree. Matrix operations are implemented in the scipy sparse-matrix package, and the "unrolling" code performs a number of optimizations to the sequence in-line: one important one is to use the fact that $\mathbf{v}_X \circ (\mathbf{v}_Y \mathbf{M}_{\texttt{any}}) = \mathbf{v}_X \|\mathbf{v}_Y\|_1$ to avoid explicitly building $\mathbf{M}_{\texttt{any}}$.

4 Related Work

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 may lead to a useful architecture [15] or regularizer [12].

Recently [11] 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 (similar to [18]), allowing structure learning as well. However, published experiments with the system been limited to very small datasets. Another recent paper [1] describes a system in which non-logical but compositionally defined expressions are converted to neural components for question-answering tasks.

Explicitly grounded probabilistic first-order languages. Many first-order probabilistic models are implemented by "grounding", i.e., conversion to a more traditional representation.[1] For example, Markov logic networks (MLNs) are a widely-used probabilistic first-order model [10] 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), [2] a MLN variant which uses a more tractible hinge loss, as well as lifted relational neural networks [13], a recent model which grounds first-order theories to a neural network. However, any grounded model for a first-order theory can be very large, limiting the scalability of such techniques.

Probabilistic deductive databases and tuple independence.

TensorLog is superficially similar to the *tuple independence* model for PrDDB's [14], which use Θ to define a distribution, $\Pr(I|\mathcal{DB},\Theta)$, over "hard" databases (aka *interpretations*) I. In particular, to generate I, each fact $f \in \mathcal{DB}$ sampled by independent coin tosses, i.e., $\Pr_{\texttt{TupInd}}(I|\mathcal{DB},\Theta) \equiv \prod_{t\in I}\theta_t \cdot \prod_{t\in\mathcal{DB}-I}(1-\theta_t)$. The probability of a derived fact f is defined as follows, where $[\![\cdot]\!]$ is a zero-one indicator function:

$$\Pr_{\texttt{TupInd}}(f|\mathcal{T},\mathcal{DB},\Theta) \equiv \sum_I [\![f \in \mathit{Model}(I,\mathcal{T})]\!] \cdot \Pr(I|\mathcal{DB},\Theta) \tag{3}$$

There is a large literature (for surveys, see **(author?)** [14, 4]) on approaches to tractibly estimating Eq 3, which naively requires marginalizing over all $2^{|\mathcal{DB}|}$ interpretations. One approach, taken by the ProbLog system [5], relies on the notion of an *explanation*. An *explanation* E for f is a *minimal* interpretation that supports f: i.e., $f \in \mathit{Model}(\mathcal{T},E)$ but $f \notin \mathit{Model}(\mathcal{T},E')$ for all $E' \subset E$. It is easy to show that if $E'' \supset E$ then $f \in \mathit{Model}(\mathcal{T},E'')$; hence, the set $\mathcal{E}x(f)$ of all explanations for f is a more concise representation of the interpretations that support f.

Under the tuple independence model, the marginal probability of drawing some interpretation $I \supseteq E$ is simply

$$\sum_{I\supseteq E}\prod_{f'\in I}\theta_{f'}\prod_{f'\in\mathcal{DB}-I}(1-\theta_{f'}) = \prod_{f'\in E}\theta_{f'}$$

while in TensorLog,

$$\Pr_{\texttt{TenLog}}(f) = \frac{1}{Z}g_{\texttt{TenLog}}(f),\ \text{where } g_{\texttt{TenLog}}(f) = \sum_{E\in\mathcal{E}x(f)}\prod_{f'\in E}\theta_{f'}$$

So TensorLog's score for a single-explanation fact is the same as under $\Pr_{\texttt{TupInd}}$, but more generally only approximates Eq 3, since

$$\begin{aligned}\Pr_{\texttt{TupInd}}(f) &= \sum_I [\![f \in \mathit{Model}(I,\mathcal{T})]\!] \cdot \Pr(I) = \sum_{I:f\in\mathit{Model}(I,\mathcal{T})}\prod_{f'\in I}\theta_{f'}\prod_{f'\in\mathcal{DB}-I}(1-\theta_{f'})\\ &= \sum_{E\in\mathcal{E}x(I)}\sum_{I\supseteq E}\prod_{f'\in I}\theta_{f'} \neq \sum_{E\in\mathcal{E}x(f)}\prod_{f'\in E}\theta_{f'} = g_{\texttt{TenLog}}(f)\end{aligned}$$

the inequality occurring because TensorLog overcounts interpretations I that are supersets of more than one explanation.

This approximation step is important to TensorLog's efficiency, however. Exact computation of probabilities in the tuple independence model are #P hard to compute [5] *in the size of the set of*

[1] For a survey of such models see [7].

Table 1: Comparison of TensorLog to ProbLog2 and ProPPR

	Social Influence		Path-finding	
ProbLog2	20 nodes	40-50 sec	16x16 grid, $d = 10$	100-120 sec
TensorLog	3327 nodes	0.84 msec	16x16 grid, $d = 16$	0.061 msec
			64x64 grid, $d = 64$	0.103 msec

	ProPPR		TensorLog	
	AUC	sec	AUC	sec
Cora (13k facts,10 rules)	97.6	97.9s	83.2	102.8s
Wordnet (276k facts)				
Hypernym (46 rules)	93.4	166.8s	93.3	154.9s
Hyponym (46 rules)	92.1	165.6s	92.8	152.5s
Deriv. Related (49 rules)	8.2	166.6s	6.7	168.2s
Freebase15k (923k facts)				
division-2nd-level	56.4	128.5s	50.8	95.7s
person-profession	45.8	24.4s	50.0	13.7s
actor-performance	37.4	19.0s	38.0	13.7s

explanations, which as noted, can itself be exponentially large. A number of methods have been developed for approximating this computation, or performing it as efficiently as can be done—for example, by grounding to a boolean formula and converting to a decision-tree like format that facilitates counting [14]. Below we experimentally compare inference times to ProbLog2, one system which adopts these semantics.

Stochastic logic programs and ProPPR. TensorLog is more closely related to stochastic logic programs (SLPs) [3]. 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 assocated probability θ_r. Let $N(r, E)$ be the number of times r was used in deriving the explanation E: then in SLPs, $\Pr_{\texttt{SLP}}(f) = \frac{1}{Z} \sum_{E \in \mathcal{E}x(f)} \prod_r \theta_r^{N(r,E)}$. 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 $\theta_{\texttt{weighted(r)}} = \theta'_{\texttt{r}}$, where Θ' is the parameters for a SLP.

SLPs can be *normalized* or *unnormalized*; in normalized SLPs, Θ is defined so 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 [3].

ProPPR [17] 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 size linear in m, α^{-1}, ϵ^{-1}, and where m is the number of training examples, α is the reset parameter, deg_{itmax} is the maximum degree of the proof graph, and ϵ is the pointwise error of the approximation. Asymptotic analysis suggests that ProPPR should be faster for very large database 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.

5 Experiments

We compared TensorLog's inference time with ProbLog2, a mature probabilistic logic programming system which implements the tuple independence semantics, on two inference problems described in [5]. One is a version of the "friends and smokers" problem, a toy model of social influence.

In [5] small graphs were artificially generated using a preferential attachment model, the details of which were not described; instead we used a small existing network dataset[2] which displays preferential-attachment statistics. The inference times we report are for the same inference tasks, for a subset of 120 randomly-selected entities. In spite of querying six times as many entities, TensorLog is many times faster. We also compare on a path-finding task, also described in [5], which is intended to test performance on deeply recursive tasks. Again TensorLog shows much faster performance, and better scalability[3]; we note however that ProbLog2 implements a much more expressive logic than considered here.

We also compared experimentally with ProPPR on several tasks. One was a citation-matching task (from [17]), in which ProPPR was favorable compared to MLNs[4]. Motivated by recent comparisons between ProPPR and embedding-based approaches to knowledge-base completion [16], we also compared to ProPPR on six relation-prediction tasks[5] involving two databases, Wordnet and FreeBase15k, a 15,000-entity subset of FreeBase, using rules from the (non-recursive) theory used in [16].

In all of these tasks parameters are learned on a separate training set. For TensorLog's learner, we optimized unregularized cross-entropy loss, using a fixed-rate gradient descent learner. We set the learning rate to 0.1, used no regularization, and used a fixed number of epochs (30), which approximately matched ProPPR's learning time.[6] The parameters θ_f are simply "clipped" to prevent them becoming negative (as in a rectified linear unit) and we use softmax to convert the output of the g^p functions to distributions. We used the default parameters for ProPPR's learning.

Encouragingly, the accuracy of the two systems after learning is comparable, even with TensorLog's rather simplistic learning scheme. ProPPR, of course, is not well suited to tight integration with deep learners.

6 Concluding Remarks

Large knowledge bases (KBs) are useful in many tasks, but integrating this knowledge into deep learners is a challenge. To address this problem, we described a probabilistic deductive database, called TensorLog, in which reasoning is performed with a differentiable process. The current TensorLog prototype is limited in many respects: for instance, it is not multithreaded, and only the simplest learning algorithms have been tested. In spite of this, it appears to be comparable to more mature first-order probabilistic learners in learning performance and inference time—while holding the promise of allowing large KBs to be tightly integrated with deep learning.

Acknowledgements

Thanks to William Wang for providing some of the datasets used here. The authors would like to thank numerous colleagues for helpful discussions, but won't, because this submission is anonymous and that would be a dead giveaway. We are also grateful to our funders, but of course, we won't tell you who they are. You might try and guess, but that would be cheating.

[2]The Citeseer dataset from [8].

[3]Unlike ProbLog2, our system performs recursion to a fixed maximum depth, which we set to 99 for these tasks.

[4]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

[5]We chose this protocol since the current TensorLog implementation can only learn parameters for one target relation at a time.

[6]Since the current TensorLog implementation is single-threaded we used only one thread for ProPPR as well.

References

[1] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Learning to compose neural networks for question answering. *CoRR*, abs/1601.01705, 2016.

[2] Matthias Brocheler, Lilyana Mihalkova, and Lise Getoor. Probabilistic similarity logic. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 2010.

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

[4] Luc De Raedt and Kristian Kersting. *Probabilistic inductive logic programming*. Springer, 2008.

[5] Daan Fierens, Guy Van Den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. To appear in Theory and Practice of Logic Programming, 2016.

[6] Matthew R. Gormley, Mark Dredze, and Jason Eisner. Approximation-aware dependency parsing by belief propagation. *Transactions of the Association for Computational Linguistics (TACL)*, 2015.

[7] Angelika Kimmig, Lilyana Mihalkova, and Lise Getoor. Lifted graphical models: a survey. *Machine Learning*, 99(1):1–45, 2015.

[8] Frank Lin and William W. Cohen. Semi-supervised classification of network data using very few labels. In Nasrullah Memon and Reda Alhajj, editors, *ASONAM*, pages 192–199. IEEE Computer Society, 2010.

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

[10] Matthew Richardson and Pedro Domingos. Markov logic networks. *Mach. Learn.*, 62(1-2):107–136, 2006.

[11] Tim Rocktäschel and Sebastian Riedel. Learning knowledge base inference with neural theorem provers. In *NAACL Workshop on Automated Knowledge Base Construction (AKBC)*, 2016.

[12] Tim Rocktäschel, Sameer Singh, and Sebastian Riedel. Injecting logical background knowledge into embeddings for relation extraction. In *Proc. of ACL/HLT*, 2015.

[13] Gustav Sourek, Vojtech Aschenbrenner, Filip Zelezný, and Ondrej Kuzelka. Lifted relational neural networks. *CoRR*, abs/1508.05128, 2015.

[14] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. Probabilistic databases. *Synthesis Lectures on Data Management*, 3(2):1–180, 2011.

[15] Geoffrey Towell, Jude Shavlik, and Michiel Noordewier. Refinement of approximate domain theories by knowledge-based artificial neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, Massachusetts, 1990. MIT Press.

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

[17] William Yang Wang, Kathryn Mazaitis, and William W Cohen. 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*, pages 2129–2138. ACM, 2013.

[18] William Yang Wang, Kathryn Mazaitis, and William W Cohen. Structure learning via parameter learning. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 1199–1208. ACM, 2014.