Distributed Big Data Analytics Project: Tensorlog - A Differentiable Database

Priyansh Trivedi[1] and Gaurav Maheshwari[2]

[1] Informatik IV, Universität Bonn, Germany
`priyansh.trivedi@uni-bonn.de`
[2] Informatik IV, Universität Bonn, Germany
`gaurav.maheshwari@uni-bonn.de`

Project Repo : `https://github.com/saist1993/tensor-related-activities`

1 Introduction

With the increasing complex and wide variety of data, the problem of learning rules over the structured and unstructured data has attracted significant research efforts [12]. There's a clear commercial incentive in learning first order logic (FOL) rules[1] as apart from providing insights and patterns on the data, they are human-interpretable as well. These rule are also useful representation for knowledge base tasks [6], such as link prediction, entity resolution etc . Often the underlying logic of the rules are probabilistic in nature, as is the case with rules learned by Markov Logic Networks [8] or ProPPr [11].

One of the major challenge is to develop systems which can integrate and evaluate these probabilistic rules and provide a query mechanism over the data. In this direction, we present a distributed implementation of TensorLog [3] a probabilistic deductive database which employs a differntiable process for reasoning. TensorLog converts each rule and query into a differentiable function in neural network infrastructure [4]. Having an end-to-end differntiable query evaluation system also leads to a close integrating with a differentiable rule learning systems and thus enables us to jointly train them. They also provide a mechanism to integrate large scale knowledge base into the "gradient" based deep learning system. Following are the contribution to the lab

- Implementation of scalable TensorLog in Python using shared matrices in Theano [9].
- Semi complete implementation of distributed version of TensorLog in Scala using Spark[13] and Mxnet[2].
- Evaluation of both the system on smoker-cancer data set [1].

The article is organized into the following sections: (2)Problem Definition, where the overview of the problem is described, (3) approach of the system is described ,(4) in which the Architectural Pipeline is elaborated, (4)Evaluation , where various evaluation criteria are discussed, and (5) Time line of the project.

2 Problem Definition

In this project we aim to find the distribution of $\mathbf{Y}$ over the database DB, given

[1] https://github.com/boost-starai/BoostSRL/wiki/Toy-Cancer-Dataset

- A query of form $p(\text{x},\mathbf{Y})$
- A DB with set of ground facts, where each facts is represented using a unary or binary relationship.
- A set of rules, in the form of horn clauses, each with a associated probability.

The whole process of reasoning should be completely differentiable with inferencing in linear time with respect to database size. The following section describes the approach used by the system.

3 Approach

3.1 Background

A database, DB, is a set $\{f_1, ...f_N\}$ 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 C. A theory, T, is a set of function-free Horn clauses. Rules are written $A : -B_1, ..., B_k$, where A is called the *head* of the clause, $B_1, ..., B_k$ is the body, and A and the B_iś are called literals. Rules can be understood as logical implications.

To introduce soft computation into this model, a parameter vector Θ is added, which associates each fact $f \in DB$ with a positive scalar θ_f (as shown in the example). The semantics of this parameter vary in different Probabilistic Deductive Database (PrDDB) models, but Θ will always define a distribution $Pr(f|T, DB, \Theta)$. These formalisms have been traditionally followed in other scientific articles regarding PrDDBs.

3.2 Approach Overview

In this section, we discuss the manner in which we model the problem of finding answers to a particular family of queries by a series of numeric functions. In [3], a constant $c \in C$ is represented as $\boldsymbol{u_c}$, a one-hot row-vector. Similarly, a binary predicate p is represented by a sparse matrix $\boldsymbol{M_p}$ (in the case of unary predicates, $\boldsymbol{M_p}$ is a sparse row vector instead).

In PrDDBs, queries solve the dual purpose of (i) retrieving answers for a desired expression, and (ii) verifying if a particular fact is derivable. We focus on the second type of queries, hereby referred to as *Argument Retrieval Queries*, and represented as $p(c, Y)$. These queries are expected to fetch responses, which are modelled as a conditional distribution over possible substitutions of Y, encoded as vector $v_{Y|c}$, such that for all constants $d \in C$,

$$v_{Y|c}[d] = Pr\left(f = p\left(c, d\right) \mid f \in U_{p(c,y)}, T, DB, \Theta\right) \tag{1}$$

where, $U_{p(c|Y)}$ is the set of facts f that match *p(c, Y)*. Further, a predicate symbol f^p_{io} denotes a *query response function* for all queries with predicate p and node io, i.e. queries of the form *p(c, Y)*. When given a one-hot encoding of c, this function returns the appropriate conditional probability vector:

$$f^p_{io}(\mathbf{u}_c) \equiv \mathbf{v}_{Y|X} \ where \ \forall d \in C : \mathbf{v}_{Y|c}[d] = Pr(f = p(c, d)|f \in U_{p(c,Y)}, T, DB, \Theta) \tag{2}$$

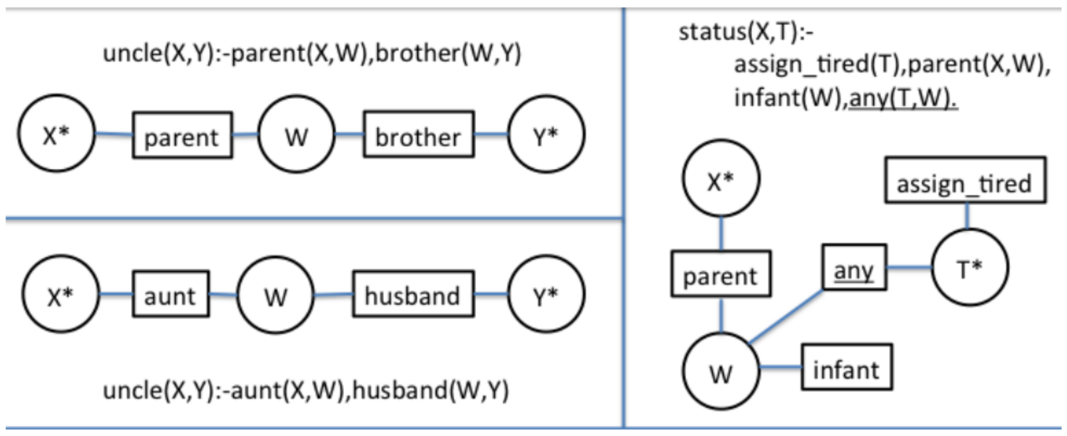
uncle(X,Y):-parent(X,W),brother(W,Y)
X*
parent
W
brother
Y*
X*
aunt
W
husband
Y*
uncle(X,Y):-aunt(X,W),husband(W,Y)
status(X,T):-
assign_tired(T),parent(X,W),
infant(W),any(T,W).
X*
assign_tired
parent
any
T*
W
infant

Figure 1: Some examples of Factor Graphs representing a different Datalog rule.

3.3 Belief Propagation

In this section, we describe the process of estimating the probability defined in equation 2, which is expected to return the desired distribution, representing the query $p(c, Y)$ (also referred to as the target rule r).

Firstly, a factor graph G_r is constructed, representing the rule body of r. For every variable W in the rule body, G_r will contain a random variable node W. And for every literal $q(W_i, W_j)$, there exists a factor with potential $\boldsymbol{M_q}$ between nodes W_i, and W_j. Figure 1 contains few examples of such factor graphs.

Thereafter, we use the algorithm sketched in Figure 2 over G_r, as proposed in [7]. The algorithm, which is a graph based message passing algorithm, consists of two mutually recursive routines, and is invoked by requesting a message from the output variable to a fictional output literal. Upon invoking it in this manner, the algorithm returns a differentiable mathematical expression establishing a relation between the desired marginal distribution f_{io}^{p} (over the different constants in the KB), $\boldsymbol{u_c}$ (representing c of r), $\boldsymbol{M_{q_i}}$ (representing the predicate literals in the rule body of r). This expression is finally normalized, in order for it to be a probabilistic distribution.

4 Implementation

In this section, we describe the ways in which we implement the system proposed in Section 3. We have implemented two versions of the system, using different programming languages, paradigms, and numerical optimization libraries. The reasons of the same will be explained in the subsection 4.2. Broadly, both implementations can be divided in three parts: (i) data pre-processing, (ii) belief propagation, (iii) training. The following section describes each step in detail.

```
define compileMessage($L \to X$):
    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($X_i \to L$)
        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($X_i \to L$)
        emitOperation( $\mathbf{v}_{L,X} = \mathbf{v}_o \cdot \mathbf{M}_p^T$ )
    return $\mathbf{v}_{L,X}$
```

```
define compileMessage($X \to L$):
    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($L_i \to X$)
        emitOperation($\mathbf{v}_X = \mathbf{v}_1 \circ \cdots \circ \mathbf{v}_k$)
        return $\mathbf{v}_X$
```

Figure 2: Algorithm for unrolling belief propagation on a polytree into a sequence of message-computation operations.

4.1 Implementation Overview

Before the system can evaluate queries, it needs to be trained on the target *KB*. To do so, three characteristic data is required of each Knowledge base: (i) a list of all the triples/facts in the KB, (ii) all the *rule templates* that the system is expected to evaluate, and (iii) training examples consisting of rules of the form $p(c, Y)$ and its corresponding answers. Here, by *rule templates*, we refer to rules of the form $p(X, Y)$ (as opposed to rules of the form $p(c, Y)$) i.e. rules with no constants in the rule head.

The matrices $\boldsymbol{M_{q_i}}$ corresponding to every predicate q_i are prepared in the first step. Thereafter, we make a factor graph G_r for every *rule template* of rule r_j, and collect the mathematical expression corresponding to it, in the form of a function $g_{io}^{r_j}$. This function takes the one-hot representation of c, the input constant in a rule head, and outputs a distribution over all the entities in the KB. The M_p matrices present in rule body in r_j, are the parameters of each function that we intend to optimize. Finally, we use the examples to train each function, and learn its parameters.

Thereafter, the model can evaluate any rule r_{eval}, as long as (i) the system has been trained on the *kind* of rules, to which r_{eval} belongs, or (ii) the BP over $G_{r_{eval}}$ returns an expression comprising solely of pretrained M_{q_i}s.

4.2 Two different Implementations

In the initial phase of the project we had research as well as technical debt and those choose Python as the principal language for implementation. Python was also pivotal as it has a very mature numerical optimization library, which supported sparse matrix. Since the aim is to integrate the project in SANSA[2], we have started an implementation of the distributed version of TensorLog using Scala and Spark. The implementation is still immature, and for it's limitation, see Section 6. We will explicitly state the major differences in the two implementations in following sub section.

[2]http://sansa-stack.net/

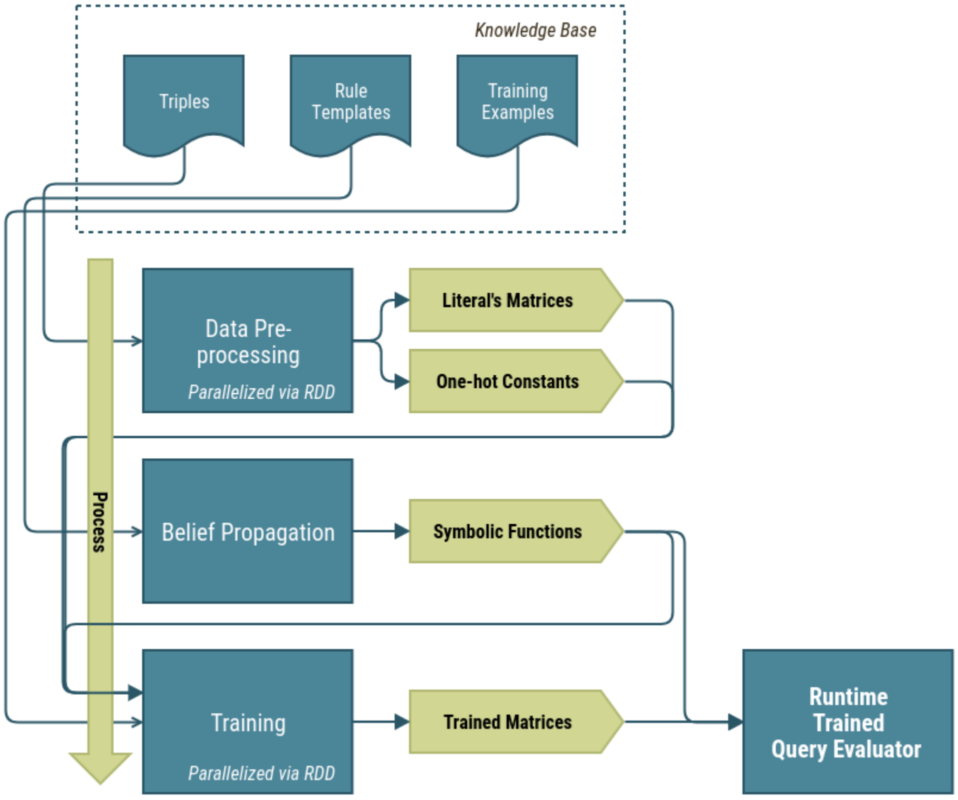

Figure 3: An overview of the system architecture.

4.3 Pre-processing

In the first phase of the process, we use the list of all the facts to create a one-hot row vector $\boldsymbol{u_c}$ all the constants (entities) c. We also create a sparse matrix $\boldsymbol{M_p}$ for a binary predicate p, where $M_p[a,b] = \theta_{p(a,b)}$ if $p(a,b) \in KB$, or an analogous row vector v_q for unary predicate q.

This process involves iterating over all the facts in the KB twice, where in the first iteration, every data point (constant, or factor) is label encoded. While in the python implementation, the iteration was done natively, we utilized Apache Spark's implementation of Resilient Distributed Datasets (RDDs)[] to accomplish this in Scala.

4.4 Belief Propagation

This phase of the project pertains to computing $g_{io}^{r_j}$ corresponding to all the available *rule templates*. We begin with parsing each rule r in the list of rule templates available for this KB, and creating the corresponding factor graph G_r. We then execute belief propagation on the graph to receive the corresponding differentiable mathematical expression in the form of the function $g_{io}^{r_j}$.

This function is symbolic (declarative) in nature, which is to suggest that none of its parameters M_q, or the input u_c has a concrete value at this time. This symbolic nature enables us to compute gradients on it automatically at the time of execution. The symbols and operations over them are defined in Theano for the python implementation, and Mxnet for Scala.

4.5 Training

Finally, we use the training examples file to train the functions created in Section 4.4. We represent the results of a query r_{eval} in this file by a row-vector y_{true} defined as: $y_{true}[i] = \theta_i$, where θ_i is the probability of that constant being retrieved as a valid answer of r_{eval}. y_{true} is used to estimate the error of the function which itself outputs another distribution over the constants. Since our function $g_{io}^{r_{eval}}$ is differentiable, we use gradient descent[] to optimize the parameters of the model (matrices M_q representing the literals in the rule head of r_eval).

It is to be noted that there is a many-many correspondence between the matrices and the functions, thus, in optimizing a single function (via gradient descent), we affect multiple matrices, and therefore, the system can evaluate queries based on unseen *rule templates* as long as its body contains no new literal.

Finally, while the python implementation of this is single threaded, Spark enables us to parallelize the training process and reduce the training time significantly. At the time of writing this report, we're still exploring ways to asynchronously update the matrices M_p.

5 Evaluation

The aim of the evaluation was to understand the scalability and accuracy of the system. We used two datasets namely:

- Friends and smokers : A toy data set which models the social influences as described in [5]. We evaluated our Scala implementation on this dataset and measured the average time required for the evaluating of the queries.
- Wordnet : This data set consists of 276k facts with rules generated using [10]. We used AUC on relation prediction task as a metric for the dataset. Python based implementation was used here as the task requires construction of sparse matrix.

In all the task parameters are learned on separate training set, with initial learning rate as 0.1, fixed epochs (30) and no regularization. On the Wordnet task we achieved an AUC of 81.28 on the prediction of Hyponym. TensorLog reports an AUC of 92.1, which we believe can be achieved by hyper-parameter tuning. One the Friends and smokers dataset, the average time for evaluation of the queries is **??**[3].

6 Limitation and Future Work

The current implementation in Scala uses Mxnet, which does not support sparse matrices, rendering it impractical for large datasets. Also Scala implementation does not rely on the sparlk-Mxnet integration library, which can help in distributing the training and thus improving the training speed. Due to immature library and technical debt we choose not to use it, but plan to integrate it in future. The current implementation in Python is also not multi-threaded and is database specific.

[3]The training is still in the process

7 Time Line

- Week 1 (23 May) - Background research for problem statment.
- Week 2 - Finalizing project statement and understand the Problem.
- Week 3 - Understanding Theano and basic prototyping in Python.
- Week 4 - Re-implementing the BP equation using shared Matrices and bug solving.
- Week 5 - Completion of Python based system and it's Evaluation
- Week 6 - Understanding Mxnet and spark
- Week 7 - Basic factor graph generation, BP equation and parsers for the data set
- Week 8 - Evaluation and Training of scala. Lab report.

In all the task both authors contributed equally.

References

[1] CL Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 275:314–347, 2014.

[2] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[3] William W Cohen. Tensorlog: A differentiable deductive database. *arXiv preprint arXiv:1605.06523*, 2016.

[4] William W Cohen, Fan Yang, and Kathryn Rivard Mazaitis. Tensorlog: Deep learning meets probabilistic dbs. *arXiv preprint arXiv:1707.05390*, 2017.

[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. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.

[6] Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. Amie: association rule mining under incomplete evidence in ontological knowledge bases. In *Proceedings of the 22nd international conference on World Wide Web*, pages 413–422. ACM, 2013.

[7] Matthew R Gormley, Mark Dredze, and Jason Eisner. Approximation-aware dependency parsing by belief propagation. *arXiv preprint arXiv:1508.02375*, 2015.

[8] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine learning*, 62(1):107–136, 2006.

[9] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[10] William Yang Wang and William W Cohen. Learning first-order logic embeddings via matrix factorization. In *IJCAI*, pages 2132–2138, 2016.

[11] 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 Information & Knowledge Management*, pages 2129–2138. ACM, 2013.

[12] Fan Yang, Zhilin Yang, and William W Cohen. Differentiable learning of logical rules for knowledge base completion. *arXiv preprint arXiv:1702.08367*, 2017.

[13] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.