MLog: Towards Declarative In-Database Machine Learning

Xupeng Li† Bin Cui† Yiru Chen† Wentao Wu* Ce Zhang‡

†School of EECS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University {lixupeng, bin.cui, chen1ru}@pku.edu.cn
*Microsoft Research, Redmond wentao.wu@microsoft.com
‡ETH Zurich ce.zhang@inf.ethz.ch

ABSTRACT

We demonstrate MLOG, a high-level language that integrates machine learning into data management systems. Unlike existing machine learning frameworks (e.g., TensorFlow, Theano, and Caffe), MLOG is declarative, in the sense that the system manages all data movement, data persistency, and machine-learning related optimizations (such as data batching) automatically. Our interactive demonstration will show audience how this is achieved based on the novel notion of *tensoral views* (TViews), which are similar to relational views but operate over tensors with linear algebra. With MLOG, users can succinctly specify not only simple models such as SVM (in just two lines), but also sophisticated deep learning models that are not supported by existing in-database analytics systems (e.g., MADlib, PAL, and SciDB), as *a series of cascaded TViews*. Given the declarative nature of MLOG, we further demonstrate how query/program optimization techniques can be leveraged to translate MLOG programs into native TensorFlow programs. The performance of the automatically generated TensorFlow programs is comparable to that of hand-optimized ones.

1. INTRODUCTION

As of 2016, it is no longer easy to argue against the importance of supporting machine learning in data systems. In fact, most modern data management systems support certain types of machine learning and analytics. Notable examples include MADlib, SAP PAL, MLlib, and SciDB. These systems are tightly integrated into the relational data model, but treat machine learning as black-box functions over relations/tensors. This results in a lack of *flexibility* in the types of machine learning models that can be supported.

On the other hand, machine learning systems such as TensorFlow, Theano, and Caffe are made much more expressive and flexible by exposing the mathematical structure of machine learning models to the users. However, these systems do not have a *declarative* data management layer — it is the user's responsibility to deal with such tedious and often error-prone tasks as data loading, movement, and batching in these systems.

Given these limitations, it is not surprising that higher-level machine learning libraries such as Keras have become increasingly popular. Given the data as a `numpy` array, Keras is fully declarative and users only need to specify the logical dependencies between these arrays, rather than specifying how to solve the resulting machine learning model. However, a system like Keras does not provide data management for these `numpy` arrays and it is still the user's responsibility to take care of low-level programming details such as whether these arrays fit in memory, or textbook database functionality such as reuse computation across runs or "time travel" through training snapshots. Moreover, Keras cannot be integrated into the standard relational database ecosystem that hosts most of the enterprise data.

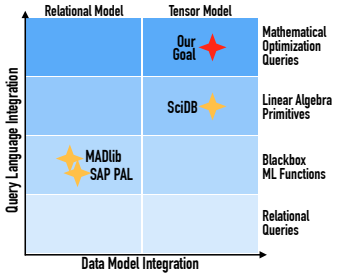


Figure 1: Schematic Comparison with Existing Systems.

In this paper, we demonstrate MLOG, a system that aims for marrying Keras-like declarative machine learning to SciDB-like declarative data management. In MLOG, we build upon a standard data model similar to SciDB, to avoid neglecting and reinventing decades of study of data management. Our approach is to extend the query language over the SciDB data model to allow users to specify machine learning models in a way similar to traditional relational views and relational queries. Specifically, we demonstrate the following three main respects of MLOG:

- **(Declarative Query Language)** We demonstrate a novel query language based on *tensoral views* that has formal semantics compatible with existing relational-style data models and query languages. We also demonstrate how this language allows users to specify a range of machine learning models, including deep neural networks, very succinctly.
- **(Automated Query Optimization)** We demonstrate how to automatically compile MLOG programs into native TensorFlow programs using textbook static analysis techniques.
- **(Performance)** We demonstrate the performance of automatically generated TensorFlow programs on a range of machine learning tasks. We show that the performance of these programs is often comparable to (up to 2× slower than) manually optimized TensorFlow programs.

Limitations. As a preliminary demonstration of our system, the current version of MLOG has the following limitations. First, our ultimate goal is to fully integrate MLOG into SciDB and Post-

Copyright 2017 VLDB Endowment 2150-8097/17/08.

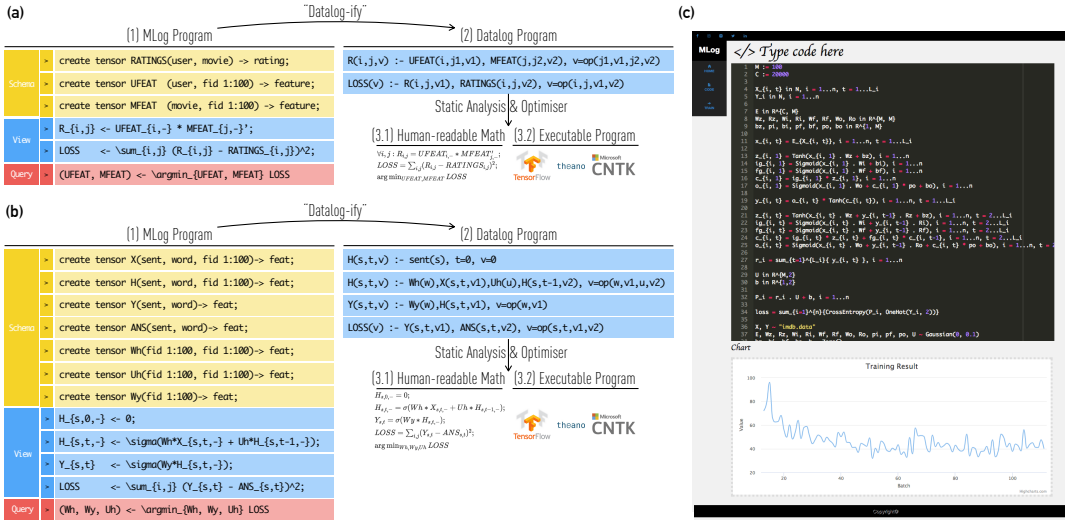


Figure 2: MLOG Examples for (a) Matrix Factorization and (b) Recurrent Neural Network. (c) User Interface of MLOG.

greSQL such that it runs on the same data representation and users can use a mix of MLOG and SQL statements. Although our formal model provides a principled way for this integration, this functionality has not been implemented yet. In this demonstration, we focus on the machine learning component and leave the full integration as future work. Second, the current MLOG optimizer does not conduct special optimizations for sparse tensors. (In fact, it does not even know the sparsity of the tensor.) Sparse tensor optimization is important for a range of machine learning tasks, and we will support it in the near future. Third, the current performance of MLOG can still be up to $2\times$ slower than hand-optimized TensorFlow programs. It is our ongoing work to add more optimization rules into the optimizer and to build our system on a more flexible backend, e.g. Angel [6, 5]. Despite these limitations, we believe the current MLOG prototype can stimulate discussions with the audience about the ongoing trend of supporting machine learning in data management systems.

Reproducibility and Public Release. The online demo will be up before the conference. For now, all MLOG programs, handcrafted TensorFlow programs, and automatically generated TensorFlow programs are available at `github.com/DS3Lab/MLog`.

2. THE MLOG LANGUAGE

In this section, we present basics for the audience to understand the syntax and semantics of the MLOG language.

2.1 Data Model

The data model of MLOG is based on *tensors*–all data in MLOG are *tensors* and all operations are a subset of linear algebra over tensors. In MLOG, the tensors are closely related to the relational model; in fact, *logically*, a tensor is defined as a special type of relation. Let T be a tensor of dimension $dim(T)$ and let the index of each dimension j range from $\{1, ..., dom(T, j)\}$. *Logically*, each tensor T corresponds to a relation $\mathcal{R}[\![T]\!]$ with $dim(T)+1$ attributes $(\underline{a_1}, ..., \underline{a_{dim(T)}}, v)$, where the domain of a_j is $\{1, ..., dom(T, j)\}$ and the domain of v is the real space $\mathbb{R}$. Given a tensor T,

$$\mathcal{R}[\![T]\!] = \{(a_1, ..., a_{dim(T)}, v) | T[a_1, ..., a_{dim(T)}] = v\},$$

where $T[a_1, ..., a_{dim(T)}]$ is the tensor indexing operation that gets the value at location $(a_1, ..., a_{dim(T)})$.

Algebra over Tensors. We define a simple algebra over tensors and define its semantics with respect to $\mathcal{R}[\![-]\!]$, which allows us to tightly integrate operation over tensors into a relational database and Spark with unified semantics. This algebra is very similar to DataCube with extensions that support linear algebra operations. We illustrate it with two example operators:

1. **Slicing σ.** The operator $\sigma_{\bar{x}}(T)$ subselects part of the input tensor and produces a new "subtensor." The j-th element of $\bar{x}$, i.e., $\bar{x}_j \in 2^{\{1,...,dom(T,j)\}}$, defines the subselection on dimension j. The semantic of this operator is defined as $\mathcal{R}[\![\sigma_{\bar{x}}(T)]\!] =$

$$\{(a_1, ..., a_{dim(T)}, v) | a_j \in \bar{x}_j \wedge (a_1, ..., a_{dim(T)}, v) \in \mathcal{R}[\![T]\!]\}.$$

2. **Linear algebra operators op.** We support a range of linear algebra operators, including matrix multiplication and convolution. These operators all have the form $\mathrm{op}(T_1, T_2)$ and their semantics are defined as $\mathcal{R}[\![\mathrm{op}(T_1, T_2)]\!] =$

$$\{(a_1, ..., a_{dim(T)}, v) | \mathrm{op}(T_1, T_2)[a_1, ..., a_{dim(T)}] = v\}.$$

2.2 MLog Program

An MLOG program Π consists of a set of *TRules* (*tensoral rules*).

TRule. Each TRule is of the form

$$T(\bar{x}) : -\mathrm{op}\left(T_1(\bar{x}_1), ..., T_n(\bar{x}_n)\right),$$

where $n \geq 0$. Similar to Datalog, we call $T(\bar{x})$ the *head* of the rule, and $T_1(\bar{x}_1), ..., T_n(\bar{x}_n)$ the *body* of the rule. We call op the *operator* of the rule. Each $\bar{x}_i$ and $\bar{x}$ specifies a subselection that can be used by the slicing operator σ. To simplify notation, we use "–" to donate the whole domain of each dimension–for example, if $\bar{x} = (5, -)$, $\sigma_{\bar{x}}(T)$ returns a subtensor that contains the entire fifth row of T. We define the *forward evaluation* of a TRule as the process that takes as input the current instances of the body tensors, and outputs an assignment for the head tensor by evaluating op.

Semantics. Similar to Datalog programs, we can define fixed-point semantics for MLOG programs. Let $\mathcal{I}$ be a *data instance* that contains the current result of each tensor. We define the *immediate consequence* of program Π on $\mathcal{I}$ as $S_\Pi(\mathcal{I})$, which contains $\mathcal{I}$ and all forward evaluation results for each TRule. The semantic of an MLOG program is the *least fixed point* of $S_\Pi(\mathcal{I})$, i.e., $S_\Pi^\infty(\mathcal{I}) = \mathcal{I}$.

Query. Given an MLOG program Π and an initial data instance $\mathcal{I}_0$, there are two ways to query the system. The *forward query* is similar to querying a Datalog program—calculating the least fixed point of the program $S_\Pi^\infty(\mathcal{I})$. The MLOG language also supports another type of query that we call *backward query*, which we define in more detail below. A backward query consists of a tuple $(T, \bar{x}, T_1, ..., T_m)$, which returns

$$\arg\max_{T_1,...,T_m} S_\Pi^\infty(\mathcal{I} \cup T_1 ... \cup T_m).T[\bar{x}].$$

That is, it tries to find the optimal instance of $T_1, ..., T_m$ to minimize $T[\bar{x}]$ as a result of a forward query, given $T_1, ..., T_m$.

3. USER INTERACTION MODEL

Like most SQL databases, users interact with our system by executing a sequence of MLOG *statements* in a REPL or a script. Each MLOG statement can be either a standard SQL statement, a TView, an MLOG query, or an MLOG tensor construction statement. Figure 2 illustrates an example MLOG REPL session for matrix factorization and recurrent neural network.

Tensor Construction Statements. The MLOG language extends the data definition sublanguage of SQL by adding a `CREATE TENSOR` primitive that is largely consistent with SciDB's syntax and semantics. Moreover, MLOG also supports the operation that "casts" a relational table into a tensor and loads data from files.

Tensoral Views (TViews). Figure 2(b) illustrates three TViews that define the following relationships between tensors:

$$H_{s,0} = 0, \tag{1}$$

$$H_{s,t} = \sigma(Wh * X_{s,t} + Uh * H_{s,t-1}), \tag{2}$$

$$Y_{s,t} = \sigma(Wy * H_{s,t}), \tag{3}$$

which encode a standard recurrent neural network model. Each TView corresponds to one formula. In this example, there is a recursive relationship between the tensor H and itself — the value of one slice of the tensor $H_{s,t}$ depends on the value of the "previous slice" $H_{s,t-1}$. The fixed point semantics we introduced in the previous section provides well-founded semantics for this scenario.

MLOG *Queries.* Figure 2(b) also illustrates one query in MLOG that corresponds to the following optimization problem:

$$\arg\max_{Wh,Uj,Wy} LOSS,$$

where $LOSS$ is a TView defined over, among others, Wh, Uj, and Wy. This query will find the optimal instance for Wh, Uj, and Wy that maximizes the value of $LOSS$.

3.1 Query Optimization

The execution of MLOG programs follows a typical procedure in a database system: the input high-level language is first converted into a *logical plan*, and then an automated optimizer translates the logical plan into a *physical plan*. In MLOG, the logical plan is a set of TViews and the physical plan is a TensorFlow (or other backend) program. We next outline some query optimization techniques we have employed to compile MLOG programs.

Query optimization is undertaken by first translating an MLOG program into a Datalog program, a process that we call "Datalogify." Given the Datalog program, the optimizer uses a standard static analysis technique to reason about the property of the program and generate a TensorFlow program as the physical plan. We illustrate this process by using the following query in a recurrent neural network as a running example:

$$H_{s,t,-} = \sigma(Wh * X_{s,t,-} + Uh * H_{s,t-1,-}),$$

where H and X are $3D$ tensors, and Wh and Uh are $2D$ matrices.

"Datalogify". The goal of "Datalogify"-ing an MLOG program is to analyze the *data dependencies* between tensors and provide a way to optimize the execution statically *without grounding out the whole dependency graph*. During this process, each TView is translated into a conjunctive aggregate query [2]. The process is simple: for each tensor T in the rule, we replace it with its relational representation $\mathcal{R}[\![T]\!]$. We abuse notation by still using the symbol T for $\mathcal{R}[\![T]\!]$, and the "Datalogify"-ed RNN query is:

$$H(s,t,v) : -Wh(w), X(s,t,v1), Uh(u), H(s,t-1,v2),$$
$$v = \sigma(w, v1, u, v2).$$

We can infer many properties of this query by analyzing it *statically*. For example, for each s, the forward process forms a chain (because of $t-1$ and t) and the length of the chain for a given s is decided by $|\{(s,t,v1) \in X\}|$, a quantity that one can obtain with a standard database optimizer. Second, to calculate for each (s,t), the whole relation of Wh and Uh will be used. One can use this fact to estimate the communication overhead of broadcasting Wh and Uh for different execution strategies. The MLOG optimizer takes advantage of these, and we describe one example that leads to one of the most significant improvements of performance.

Batch Optimization with Pivoting Method. One important optimization for speeding up machine learning is data batching. For example, in our previous work [3], we find that deep learning systems without proper batch optimization can be up to $5\times$ slower. As a declarative language, MLOG does not expose the batching operation to the user. Therefore, it is important to automatically detect batching opportunities.

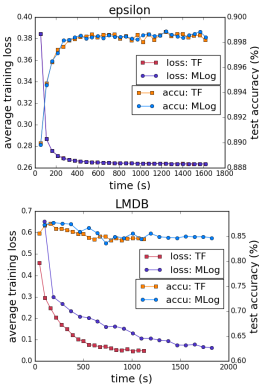
The MLOG optimizer detects batching optimization using a textbook static analysis technique for Datalog programs called the *pivoting* method [11]. The goal is to detect *connected components* in a Datalog program — components that can be evaluated in parallel without communication. For example, in the RNN query, different values of s form different connected components conditioned on Wh and Uh, which can be easily detected by the pivoting method.

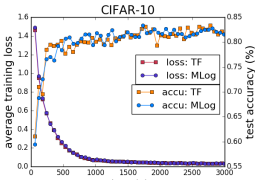
After discovering these batching opportunities, the MLOG optimizer automatically *expands* all involved n-dimensional tensors into $n+1$-dimensional tensors by adding the *batch dimension*. For example, the $3D$-tensors H and X will become $4D$-tensors in the physical plan. All operators such as matrix multiplication (e.g., $*$) will also be translated into the batched version. Note that in the above example, each chain is of a different length and zero-padding is a standard practice for batching them together — although this is not currently implemented: MLOG's optimizer has enough information to choose batching different chains together to minimize zero-padding overhead because it knows how to estimate the length of each chain with a standard cardinality estimator.

4. DEMONSTRATION SCENARIOS

Predefined Models. We demonstrate three example MLOG programs that implement three popular machine learning models—Support Vector Machines (SVM), Convolutional Neural Networks (CNN), and Long Short Term Memory networks (LSTM). These models cover a large range of machine learning applications — SVM for linear classification, CNN for image processing, and LSTM for speech and natural language processing. To the best of our knowledge, none of the existing in-database analytics systems supports all of these models.

Data Sets. We will use three public data sets and implement the three predefined models using MLOG:





Time per Batch (ms)

Dataset	MLog	TF
epsilon	16.9	16.8
CIFAR-10	119.6	118.3
LMDB	365.3	224.7

Figure 3: Experiment results on GPU

epsilon (SVM) is a binary classification dataset. The dataset has 400,000 training examples and 100,000 testing examples. Each example is a pair (x_i, y_i), where $x_i \in \mathbb{R}^{2000}$ and $y_i \in \{1, -1\}$. We implement an SVM model with hinge loss in MLog.

CIFAR-10 (CNN) is a ten-class image dataset. It contains 50,000 training and 10,000 testing images. The size of each image is $32 \times 32 \times 3$. We implement a five-layer NiN CNN [8] in MLog.

LMDB (LSTM) is a binary-class short English text paragraphs dataset. It contains 25,000 training and 25,000 testing examples. We make a vocabulary of 20,000 words with highest frequency in the training set. We then implement a one-layer LSTM model followed by logistic regression in MLog.

Scenarios. As our first scenario prepared before the presentation, we will compare the performance of the automatically generated TensorFlow programs (by the MLog optimizer) with that of the ones we manually optimized.[1] We will evaluate the performance on both CPU and GPU. The CPU machine contains an Intel Xeon E5530 @ 2.40GHz CPU and 70GB RAM and the GPU machine contains a Pascal TITAN X GPU and 256GB RAM. We will measure the training loss and testing accuracy.

Here we present some preliminary results that we will showcase during our presentation. Figure 3 shows the results on GPU while the CPU result is similar. We see that both MLog and TensorFlow achieve the same accuracy and loss on all three datasets. In terms of performance, both systems have comparable speed per batch or epoch for SVM and CNN. For LSTM, MLog takes 1.6× longer than TensorFlow — in TensorFlow we use an optimized LSTM layer as a whole while in MLog the full structure of the LSTM layer is written at the level of matrix multiplication. Clearly, in this case, MLog needs more optimizations to match Tensorflow's performance; after performance analysis, we find that this difference is because TensorFlow merges several matrix multiplications to one so that it can get higher GPU utilization (about 32% in MLog whereas 40% in TensorFlow). For example, AW, BW can be calculated in one matrix multiplication $[A^T B^T]^T W$ as long as A and B have the same shape. For CPUs, we get the same accuracy and performance for all three data sets.

As our second scenario, we will invite the audience to try out their own MLog programs via the user interface presented in Figure 2. Foreseeably, the audience may need some warm-up before they can write simple yet correct MLog programs. We plan to provide some guidance during this "training" phase. We will then show the automatically generated TensorFlow programs to the audience. However, before that we would like to encourage the audience to come up with their own TensorFLow programs first. (We can ease this task by choosing standard machine learning models with publicly available TensorFlow implementations.) We expect some intriguing and perhaps intensive technical questions and discussions regarding query optimization here, most likely from people with deeper expertise in TensorFlow and/or general query optimization. We finally run the automatically generated and user-provided TensorFlow programs to compare their performance.

[1] https://github.com/DS3Lab/MLog

5. RELATED WORK

Modern data systems often support libraries for analytics and machine learning. Examples include MADlib [4] for Greenplum and PostgreSQL, SAP PAL [9] for SAP HANA, ORE for Oracle databases. These libraries tightly integrate with the host data system and support traditional machine learning algorithms such as SVM or K-means. Our work is built upon these previous work, however, advocates a more flexible higher-level language that supports more sophisticated machine learning models, such as deep neural networks, inside existing data systems. SciDB [1] is a recent effort to extend relational database with data representations and operations for linear algebra. However, as far as we know there is no machine learning library existing for SciDB, and we hope MLog could fill that vacancy. There have been efforts to train linear models over joins [7, 10]. Compared with these efforts, MLog advocates a more unified data model based on tensors instead of relations and also provides a more expressive way to encode correlations among tensors. As a result, MLog is able to encode sophisticated machine learning models beyond linear models.

6. CONCLUSION

We have demonstrated MLog, a high-level declarative language that integrates machine learning into database systems. An MLog program is very similar to a SQL program but extends relational algebra over relations to linear algebra over tensors. This extension allows MLog to encode a range of machine learning models that are not supported in current data analytics systems. To optimize the performance of an MLog program, MLog contains a database-style query optimizer. In many cases, the resulting performance of automatically compiled MLog programs is comparable with hand-tuned TensorFlow programs.

Acknowledgement. Bin Cui is supported by the NSFC under Grant No. 61572039 and U1536201, 973 program under No. 2014CB340405, and Tecent Research Grant (PKU). Ce Zhang gratefully acknowledges the support from the Swiss National Science Foundation NRP 75 407540_167266, NVIDIA Corporation for its GPU donation, and Microsoft Azure for Research award program.

7. REFERENCES

[1] P. G. Brown. Overview of scidb: Large scale array storage, processing and analysis. In *SIGMOD*, 2010.

[2] S. Cohen, W. Nutt, and Y. Sagiv. Deciding equivalences among conjunctive aggregate queries. *J. ACM*, 2007.

[3] S. Hadjis, F. Abuzaid, C. Zhang, and C. Ré. Caffe Con Troll: Shallow ideas to speed up deep learning. In *DanaC*, 2015.

[4] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The madlib analytics library: Or mad skills, the sql. *Proc. VLDB Endow.*, 2012.

[5] J. Jiang, B. Cui, C. Zhang, and L. Yu. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 463–478. ACM, 2017.

[6] J. Jiang, L. Yu, J. Jiang, Y. Liu, and B. Cui. Angel: a new large-scale machine learning system. *National Science Review*, 2017.

[7] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, 2015.

[8] M. Lin, Q. Chen, and S. Yan. Network In Network. *ICLR*, 2014.

[9] J. MacGregor. *Predictive Analysis with SAP: The Comprehensive Guide*. SAP PRESS, 2013.

[10] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD*, 2016.

[11] J. Seib and G. Lausen. Parallelizing Datalog programs by generalized pivoting. In *PODS*, 1991.