An Overreaction to the Broken Machine Learning Abstraction: The ease.ml Vision

Ce Zhang[†], Wentao Wu,[‡] and Tian Li[*]
† ETH Zurich, Switzerland, ce.zhang@inf.ethz.ch
‡ Microsoft Research, USA, wentao.wu@microsoft.com
∗ Peking University, China, litianeecs@pku.edu.cn

The world that I was programming in back then has passed, and the goal now is for things to be easy, self-managing, self-organizing, self-healing, and mindless. Performance is not an issue. Simplicity is a big issue.

— Jim Gray, 2002

After hours of teaching astrophysicists TensorFlow and then see them, nevertheless, continue to struggle in the most creative way possible, we asked, *What is the point of all of these efforts?* [1]

It was a warm winter afternoon, Zurich was not gloomy at all; while Seattle was sunny as usual, and Beijing's air was crystally clear. One of the authors stormed out of a Marathon meeting with biologists, and our journey of overreaction begins. We ask, *Can we build a system that gets domain experts completely out of the machine learning loop? Can this system have exactly the same interface as linear regression, the bare minimum requirement of a scientist?*

We started trial-and-errors and discussions with domain experts, all of whom not only have a great sense of humor but also generously offered to be our "guinea pigs." After months of exploration, the architecture of our system, `ease.ml`, starts to get into shape—It is not as general as TensorFlow but not completely useless; in fact, many applications we are supporting can be built completely with `ease.ml`, and many others just need some syntax sugars. During development, we find that building `ease.ml` in the right way raises a series of technical challenges. In this paper, we describe our `ease.ml` vision, discuss each of these technical challenges, and map out our research agenda for the months and years to come.

1 INTRODUCTION

There has been little doubt that most fields of science is moving towards a more data-driven paradigm. In our short experience working with a diverse range of scientists at ETH for just six months, we already see the abundance of potential opportunities. With six on-going collaborative projects (e.g., [7]) with domain experts, it is the time to ask the question: *How can we raise the level of abstraction to get ourselves out of the loop?* Figure 1 is our answer.

[1]We are grateful to the great sense of humor shared by all of our collaborators—astrophysics, biology, and other domains—without whose generous support this vision is not possible at all. The only motivation of `ease.ml` is to make them happy, who give us the privilege to witness and support their journey of advancing their field.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HILDA'17, Chicago, IL, USA
© 2017 ACM. 978-1-4503-5029-7/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3077257.3077265

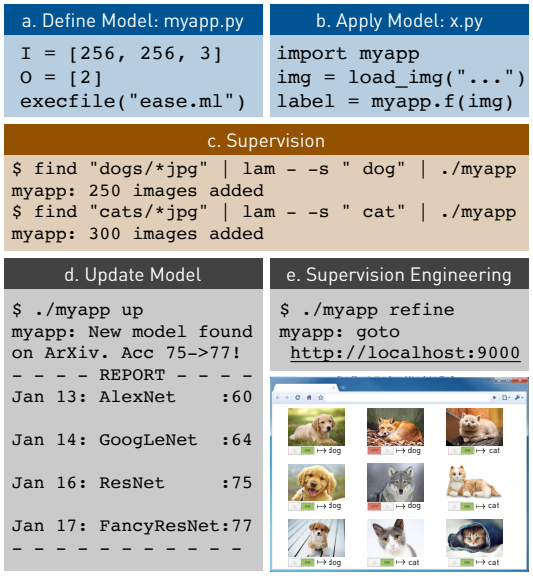

Figure 1: The design of the first version of `ease.ml`.

Users interact with `ease.ml` in five ways, and five ways only. The mental model is to think about machine learning models as an *approximator of arbitrary functions*, and nothing else:

(1) Model Definition: User defines a machine learning model in Python with the size of the input and size of the output of the function approximator s/he wants to build.

(2) Model Application: User can use the machine learning model on new inputs like any other Python functions.

(3) Supervision: Like a linear regression model, users feed an `ease.ml` application with pairs of inputs and outputs. Every time a new pair is added, the model gets automatically updated without any user intervention.

(4) Update Model: Whenever a new machine learning model is available on ArXiv (such as a new neural network model for image classification), all applications built with `ease.ml` automatically get updated.

(5) Supervision Engineering: Users manage the training corpus with a Web interface. S/he can remove training examples from the corpus. Whenever the training corpus is changed, the model gets updated automatically.

Design Rationale. The revival of deep neural network has changed the way that people view and understand the world. Machine learning systems powered by deep neural networks are often way more capable than prior state-of-the-art techniques, especially on sophisticated tasks that even perplex human beings, such as speech recognition, image classification, natural language processing, drug discovery, and so on. The emergence of the deep learning era therefore inspires not only computer scientists but also folks in various areas wherever big data is available, including astronomy, geography, oceanology, meteorology, toxicology, biomedical informatics, etc. Unfortunately, deep neural networks are such complex objects that their behavior is elusive even for top experts in this field. Building, training, and tuning performance of deep neural networks is often more of an art than science. It is thus easy to see an obvious gap here between the ever-growing demand of using deep neural networks and the pain of manually managing these networks.

In this paper, we present our vision on building a system `ease.ml` that automatically manages the process of building, training, and tuning deep neural networks. Such a system can significantly ease application development on top of deep neural networks, especially for users and developers with little expertise. The experience we gain from the past half-century database system research tells us that such a system must be both *declarative* and *at scale*. However, as the target system manages the data accessed by the deep neural networks as well as the networks themselves, our context is similar but different from the classic view adopted by modern database management systems. Current database systems are good at *scaling up* (i.e., as the volume of the data increases) but not so good at *scaling out* (i.e., as the number of applications increases). By design, it is the user's responsibility to take care of the *programs* (e.g., queries, stored procedures, etc.) that run on top of the database system. The requirement of involving into this level of detail often becomes a huge burden for application developers. Unfortunately, in general this is an intractable problem given the diversity of applications. Fortunately, in the context of applications based on deep neural networks, it is possible to build a system that manages both the data and the programs, which are neural networks. Our goal is to entirely shield the complexity of deploying deep neural networks from application developers. That is, they only need to define the *input* and *output* of the *learning problem* without specifying details such as the number of hidden layers, the linkage structure, the training algorithm, etc. (Technically, there is no barrier that prevents us from including other machine learning models into `ease.ml`. However, we want to focus on deep neural networks for ease of exposition so that our application scope is clear enough.)

"Every Machine Learning Company under the Sun". As pointed out by an anonymous reviewer, it seems that "every machine learning company under the sun" is trying to "easy the ML process." This is true — 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. Higher-level machine learning libraries such as Keras have also become increasingly popular. Our vision built upon these success, however, aims at providing an even higher-level of abstraction for the user — even a system like Keras does not provide much data management functionality and it is still the user's responsibility to take care of low-level physical decisions.

2 THE PAIN: A WAR STORY

In this section, we document our experience in supporting one of our collaborators to make the argument that that the abstraction of existing machine learning systems is broken.

Kevin's Log. Our astrophysics collaborator maintains a daily log of their research, in the form of messages on their group Slack channel. The #machine_learning channel contains all online communications, in daily basis, about their effort in using TensorFlow. Their communication started on Jan. 13th, 2016. Before Sept. 1st, 2016 (when we started to collaborate with them), they accumulated more than 300 pages of communications about the problems they were facing when using TensorFlow *all by themselves*.

Protocol. We read all of their online communications before Sept. 1st, 2016 and summarize a taxonomy of problems they were facing. We only summarize here a small part of this taxonomy (i.e., communications they had from starting the install TensorFlow to get their *first* end-to-end run, which has almost unusable quality).

Results. On Jan. 27th, 2016, they finished preparing all the training data and started to train their deep learning model. They got their first end-to-end run (a test accuracy number) on March 2nd, 2016. The performance of the system was not the bottleneck. Instead, they were mainly blocked by the *usability* of the system. More interestingly, many of the usability issues they were facing are major concerns when our community builds a relational database system. We list a few in the following.

(1) **Memory management:** The first problem they were facing (Feb. 9th to 15th) was caused by that their images have a different dimension than most standard neural networks. They started with keeping the image of the same size and applied the same network. Because the input size was much larger, TensorFlow ate all the GPU memory. Worse, there were no error diagnosis messages from the system, and they ended of spending one week in understanding what happened.
(2) **ETL:** It took them two days (Feb. 7th and 8th) to translate their data, a standard format that comfortably fits into SciDB and TensorFlow. Sounding like a trivial task, the lack of a unified model between the data processing ecosystem and machine learning ecosystem does impose challenges on their side.
(3) **Code reuse:** The second problem they were facing (Feb. 15th to March 2nd, 2016) is that, although their training pipeline can run smoothly, their testing pipeline has bugs. This is logically weird because testing runs a strictly subset of operations.
(4) **Model selection:** Model selection is tricky for astrophysicists, and the discussion endures for their whole process. For reasons we do not understand, instead of starting from standard networks, they chose to design it by themselves. As a consequence, even they had a model successfully trained and evaluated on March 2nd, 2016, the accuracy hardly beats random guess.
(5) **Hyperparameter Tuning:** Hyperparameter tuning is one of the few things they handled well. They spent one day writing a script to tune hyperparameters and scheduled the run overnight. Although most of their hassles could be solved by just switching to the Adam optimizer [4], tuning hyperparameters does not cause much problem for them. (hyperparameter tuning is not unfamiliar and they do that for their daily research.)

The Closure. An anonymous reviewer was curious about whether Kevin finally gets help from us. Yes [7]. The `ease.ml` vision is motivated by our experience in helping users like Kevin.

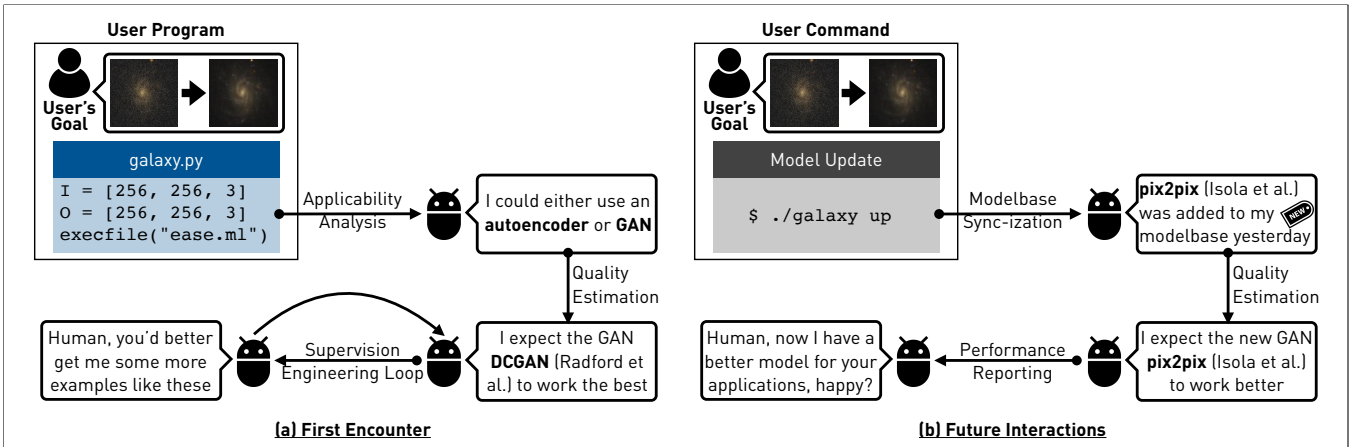

Figure 2: The interaction model of ease.ml illustrated with one of our astrophysics applications [7].

3 THE EASE.ML VISION

ease.ml focuses on a simple question—*What is the abstraction we should provide for Kevin's to make their endeavours before March 2nd, 2016 more efficient and painless?* This goal results in ease.ml's simplicity, but also results in its limitation. However, as we will see, even this simplied goal is challenging (and potentially rewarding).

Example 3.1. Figure 1 outlines the user interface of ease.ml. Users interact with ease.ml at a pretty high level: applications are expressed as a series of model invocations where each model is defined only in terms of its input and output sizes. The models are then trained with the specified datasets and can be reused for different inference tasks. The only operation that users need to train a model is *to pipe training data into* ease.ml. ease.ml automatically manages the models in the sense that it will refresh the best model found upon (i) any dataset change and (ii) any new available model (from external source).

Although the user-level code in the above example is written in Python, it highlights the general idea of having a high-level language interacting with our system that *declares* the learning task. More sophisticated applications may involve multiple, cascaded learning tasks and such a declarative language enables application developers to focus on the logical connection between these tasks without worrying about how each task is implemented.

One may now wonder if this vision is even achievable. After all, at a first glance it seems incredible, if not insane, that the user even has no idea about which models are running inside the system! To not oversimplify the problem and to convince ourselves, Figure 2 reveals more details under the hood. As shown in Figure 2(a), there are essentially two decision procedures going on. The first procedure decides which model should be used for the learning task, whereas the second procedure decides how to train the model with the given dataset. For example, suppose that ease.ml is currently equipped with the four neural networks presented in Table 1. The system needs to pick one from these candidates based on some "quality estimator." After a neural network is chosen, ease.ml then automatically trains the model with the specified input and output sizes on the given dataset. Nonetheless, for all of these to be automated, we need innovation at the system architecture level.

Specifically, unlike existing database systems with one declarative layer, ease.ml consists of two declarative layers. The first declarative layer (i.e., the top right "robot" in Figure 2(a)) maps the user-defined learning task to a neural network program specified in a declarative language that combines features of neural network computation and data manipulation. The second declarative layer (i.e., the bottom right "robot" in Figure 2(a)) takes the (declarative) neural network program returned by the first layer and generates a physical execution plan. The physical plan is finally executed by the execution engine (i.e., the bottom left "robot" in Figure 2(a)). On a first thought the second layer seems redundant. Why not just directly translate the learning problem into a physically executable program? After all, users do not need to see the intermediate declarative representation. The reason is for, as we will see later, better extensibility/scalability and query/program optimization opportunities. In a broader sense, our system is a natural extension of existing database systems: advance from *data-independent* computation to *computation-independent* application. We view our system as a first step towards this ambitious goal. Although we have mentioned that the problem is in general intractable, the inherent difficulty comes more from characterizing ubiquitous properties of applications. Within a narrower, well-defined scope, such as applications powered by deep neural networks, we do believe that computational independence is achievable.

This two-layer declarative architecture also raises a number of new challenges. First, it is now the system's, instead of the user's, responsibility to choose a neural network for a specified learning task. Akin to the classic query optimization problem in database systems, now we have a "model optimization" problem (i.e., the "quality estimator" in Figure 2(a)). Second, given that the system automatically manages both the neural networks and the data, we need a unified logical view of specifying neural network computation and data manipulation. Third, given the impedance mismatch between tensors, which are favored by neural network computation, and tables, which are favored by data manipulation, we further need a unified physical model. Should we physically treat everything in ease.ml as tensors, tables, something in between, or something else? This is not a trivial problem.

Input Size	Output Size	Applicable Family
$[A, A, B]$	$[C]$	Convolutional Neural Networks
$[A, A, B]$	$[A, A, C]$	Generative Adversial Networks
$[A, A, B]$	$[A, A, B]$	Autoencoder
$[A, \text{any}]$	$[B, \text{any}]$	Recurrent Neural Networks

Table 1: `ease.ml` automatically decides the family of neural networks that is applicable to users' application based solely on the size of inputs and the size of outputs.

4 MODEL OPTIMIZATION

A new declarative level introduces new optimization opportunities. Given the user-defined input/output specification, our system needs to come up with a neural network construction that optimizes the performance. There are two major problems here.

- *Model selection*: How to choose appropriate neural networks from available candidates?
- *Quality estimation*: How to estimate/evaluate the performance of a neural network?

A brute-force approach could easily solve the two problems altogether: train all the available neural networks on the whole training set and pick the one with the highest accuracy on the testing set. This approach has an obvious scalability issue thus only works for a handful of neural networks and small training data.

If we compare our model optimization problem with the classic query optimization problem we can see an analogy. In query optimization our task is to find the best query execution plan from a number of candidates. There we face two similar problems: (i) plan selection and (ii) plan quality estimation. To address the plan selection problem, exhaustive search is usually out of consideration except for very simple queries. Heuristics are applied to reduce the scope from which candidates are picked. (e.g., Only choose plans that are left-deep trees.) To address the plan quality estimation problem, an (often hand-crafted) cost model is built to estimate the execution overhead of the query.

Following this thought, a similar framework can be developed for the model optimization problem. However, the specifics differ and require further exploration.

4.1 Model Selection

First, the heuristics used to prune the search space are not handy. Essentially, it is not clear what kind of neural network structure is appropriate for a given learning problem, and we need some (perhaps empirical) guiding principles. So far, we are not aware of any existing work in this respect.

4.2 Quality Estimation

Second, the quality estimation respect becomes more challenging. In the context of query optimization, query plans are based on well-defined algebraic systems and thus execution cost approximation is achievable by simply following the semantics of physical operators. Unfortunately, this is not the case for neural networks, where there is no such algebraic system and performance is measured in terms of accuracy rather than execution time.[2] We therefore cannot rely on analytic approach without actually running the networks. Again, we do not consider the option of running the network on top of the whole training set, the overhead of which might be prohibitive. So the question boils down to how to predict the performance of the neural network without using all training data (or using as few data as possible). In fact, there has been a great deal of work in the literature on performance prediction for a variety of machine learning models. However, as far as we know there is little work on neural networks in this respect. Moreover, the focus of prior work has been deriving performance upper bounds rather than directly estimating performance itself. The upper bounds are usually too loose to be useful as a criterion in practice. For deep neural networks, the situation is even worse: there is no known performance upper bound. Therefore, developing practical performance prediction techniques for deep neural networks remains an untouched area and calls for more research effort.

Related Work. Quality estimation is closely related to that of the "learning curve" [2, 3], which tries to estimate the accuracy of a classifier with a *subsample* of the training data. Obviously, previous works can be used to build a baseline quality estimator. However, our quality estimator could be much more sophisticated than a learning curve estimator. Instead of just using a subsample of the data, we can use information across similar datasets or similar machine learning models. We expect that this information will significantly improve the accuracy of our estimator and finally make such an estimator much more robust in real-world systems.

5 UNIFIED LOGICAL VIEW

Having a unified logical view of neural network computation and data manipulation is not mandatory. One could simply have a system running neural networks on top of databases: just implement neural networks in Python and pull data from databases using SQL queries whenever necessary. Nonetheless, this approach has a number of drawbacks. First, relation is good for data manipulation but perhaps not good for neural network computation. As was demonstrated by TensorFlow and SciDB, array-based representations are much more efficient for such computational tasks. Not only are they natural choice with respect to the first-class citizens there (e.g., vectors, matrices, etc.), but they also render vector-oriented, GPU-based processing possible. Second, the well-understood impedance mismatch remains between Python and SQL. Third, query/program optimization is limited to the data fetching layer. As was pointed out by recent work [5], pushing down computation when fetching data can often significantly improve performance. Fourth, extensibility is poor. Whenever a new model is added into the system, a new end-to-end program has to be implemented in spite of the fact that most of the functionality can be copy-and-pasted from existing programs. Moreover, if there is an updated implementation (e.g., a better matrix multiplication algorithm), all affected programs have to be manually rewritten.

To overcome these shortcomings, in the following, we present a logical view that unifies relations and tensors (i.e., multi-dimensional arrays). Based on this unified logical view, we then briefly describe a language that combines relational algebra and linear algebra. This allows users to express neural network computation (which is essentially linear algebra) and data manipulation (which is essentially relational algebra) within one single system in a declarative manner. We further outline query/program optimization opportunities induced by taking this unified logical view. (See [6] for more details.)

[2] Of course, training time is also a concern. Nonetheless, it is not our goal to estimate training time. Rather, our goal is to estimate the "result" by training the neural network. Put it another way, we want to estimate the "query result" if we use our aforementioned analogy between query optimization and model optimization.

5.1 TViews: Union of Tensors and Relations

Logically, a tensor (i.e., a multi-dimensional array) can be 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 $\{1, ..., dom(T, j)\}$. T then 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 number $\mathbb{R}$. 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 $(a_1, ..., a_{dim(T)})$.

This logical, relational view of tensors allows us to define semantics of linear algebraic operators in terms of relations. Specifically, a linear algebraic operator op such as matrix multiplication or convolution has the uniform form $\mathsf{op}(T_1, T_2)$. Its semantic can then be defined as $\mathcal{R}[\![\mathsf{op}(T_1, T_2)]\!] =$

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

Moreover, the $\mathcal{R}[\![-]\!]$ operator also provides a natural way of manipulating tensors in relational systems — whenever a tensor T is used by a relational operator, the operator logically works over $\mathcal{R}[\![T]\!]$! Therefore, it is not difficult to conceive a system mixed with linear algebraic and relational operators, at least logically, that can manipulate tensors and relations simultaneously. We next outline such a (logical) system by presenting MLog, a Datalog-alike query language that combines relational algebra and linear algebra based on manipulating TViews.

5.2 MLog: DataLog Strikes Back

An MLog program consists of a set of *TRules* (i.e., *tensoral rules*). In the following, we first define TRule, and then present a simple example MLog program.

TRule. Each TRule is of the form

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

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 $\bar{x}_i$, as well as $\bar{x}$, specifies a subselection that can be used by the slicing operator σ defined below:

- **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 $\mathcal{R}[\![\sigma_{\bar{x}}(T)]\!] =$

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

For example, if $\bar{x} = (5, -)$, $\sigma_{\bar{x}}(T)$ returns a subtensor that contains the entire fifth row of T.[3] 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. Similarly, we can define fixed-point semantics for MLog programs.

[3]We use "–" to donate the whole domain of each dimension.

Example 5.1. The following is an MLog program with three TRules that encodes a standard recurrent neural network model:

$$H_{s,0} = 0, \quad (1)$$

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

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

Each TView (i.e., the head of each TRule) corresponds to one mathematical 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 are well defined in this scenario.

5.3 Query/Program Optimization

Query optimization is undertaken by first translating an MLog program into a Datalog program, a process that we call "Datalogify." Given the Datalog program, we then use a standard static analysis technique to reason about the property of the program, and eventually 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 where (X is the input layer, H is the hidden layer, and s is one index of the input series):

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

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

The goal of "Datalogify"-ing an MLog program is to analyze the *data dependency* among 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 [1]. The process is simple: for each tensor T in the rule, we replace it with its relational representation $\mathcal{R}[\![T]\!]$.[4] The "Datalogify"-ed RNN query is:

$$H(s, t, v) :- \quad Wh(w), X(s, t, v1), Uh(u), H(s, t-1, v2), \quad (4)$$

$$v = \sigma(w, v1, u, v2). \quad (5)$$

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 [6]. Our initial evaluation shows that the automatically generated TensorFlow programs can achieve similar performance compared with manually tuned ones [6].

6 UNIFIED PHYSICAL MODEL

Although we have demonstrated the power of having a unified logical view, our examples presented in the previous section are perhaps not so compelling given that they are pure neural network computation programs with no complicated data manipulation such as joins or nested sub-queries. While we intentionally kept those programs simple, to demonstrate the full capacity of being uniformly declarative we need a unified physical model as well. Otherwise we still do not address the (physical) impedance mismatch problem

[4]We abuse the notation by still using the symbol T for $\mathcal{R}[\![T]\!]$.

by allowing the coexistence of relations and tensors, and perhaps leave potential chance of utilizing GPU-based computation on the table, though we do have more automated query/program optimization opportunities. By a unified physical model, here we meant a single execution engine for the MLog programs. Opposite to this is a system with one relational engine and one tensor computation engine, and there is a data transformation layer whenever we need to convert relations to tensors or vice versa. Given the fact that both relational systems and tensor computation systems rely on execution plans that are directed acyclic graphs (DAGs), it is conceptually possible and tempting to have a DAG-based execution system that interleaves physical relational operators with physical tensor computation operators.

The problem is, of course, the physical data format when data is streaming through such a DAG. This problem is not important if computation is always decoupled with data retrieval: we can just generate a relational execution plan to fetch the data, convert the relational data to a tensor vector, and then generate an execution plan for tensor computation. In real world, however, it is often beneficial to push computation down to the data retrieval pipeline, as evidenced by [5], especially if data is scattered in different, normalized tables (which actually is the common case in practice). As a result, we may often have cases where computation and data manipulation are interleaved. A unified physical model can provide a consistent interface for the operators in such situations. Since TViews unify relations and tensors, it is natural to consider using one of them as the unified physical model. There are pros and cons for either of them, though, as we discuss below.

6.1 Tensors as Relations

The first thought is treating all tensors in the system as relations. This leans towards the relational view of data, which is more natural but more problematic when computation efficiency is the ultimate goal. Specifically, the linear algebraic operators now have to operate on tables rather than tensors, physically. In more detail, we always convert tensors to relations with the help of the $\mathcal{R}[\![T]\!]$ operator, even for linear algebraic operators! This is apparently an overkill.

6.2 Relations as Tensors

The other way is to do the reverse: treat all relations as tensors. This eases computation — everything is a tensor and thus not abrupt for linear algebraic operators. However, relational operators may be upset by this. One way to overcome this is to convert tensors to tables whenever we need to perform selections, joins, group-bys, etc. Nonetheless, there is a more fundamental problem here about semantics. For example, what do we mean by joining two tensors? What are the join-keys, and what is the output? Although we can logically convert a tensor to a table via the $\mathcal{R}[\![T]\!]$ operator, it does not help understand the semantics. To define the right semantics, we must trace back from the definition of the current TView to the original, relational format of the data to understand the relational semantics of the tensors behind the scenes. That is, we need to convert the TView to another relation such that the relational operation is well defined. This is not always feasible.

An even simpler example is the following. Suppose that we want to apply a filter on a TView. The filter is actually applied to the "value" column of the TView, not the addressing columns. However, our intention may not be filtering out all ineligible values. Rather, we want to filter out particular rows in the TView that correspond to, say, the "color" attribute of the "dogs" table which has been converted to a tensor for computational efficiency. Therefore, our selection condition has to be complemented by some address selection conditions, which in turn require us to keep track of the lineage of the values in the tensors so computed.

6.3 Beyond Relations and Tensors

Instead of using either relations or tensors to represent everything, is it possible to use something in between or something else? This is a difficult question which we have no good answer at this time. For performance reasons, one can further partition tensors with respect to dimensions. For example, if most of the slicing operators (defined in Section 5.2) are performed on a certain dimension, then maybe we can partition the corresponding TView into multiple TViews along that dimension. One can even try to find the best partitioning scheme for a given query/program workload. Nevertheless, the basic question here remains the same: physically, what is the commonplace and difference between a relation and a tensor? Our current understanding is that tensor is more general than relation in terms of structure — relation can be thought of as two-dimensional tensor. However, relations carry more semantics given the schematic information built in: each row is an entity and each column is an attribute of that entity. We don't see such semantics in the current definition of tensors. Perhaps, it is possible to enhance tensors with schematic information as well, which might result in a physical model that generalizes relations and tensors. We are not sure about this. As a research agenda we will start with the "relations as tensors" view and see how far we can go.

7 CONCLUSION

Our `ease.ml` vision in this outrageous paper is perhaps less bold than it actually is. Although we have built some components of the system, such as the unified logical view and its translation into a tensor-based physical plan, we are less certain about other components such as the quality estimator. Foreseeably, there are lots of challenges and we are just at the beginning of this long journey. Nonetheless, given the recent rapid progress in deep learning research and engineering, we are confident and excited as we are approaching the `ease.ml` portrait depicted in the first two figures of this paper. We hope that our overreaction to the broken abstraction of current machine learning systems that frustrated both us and the users we have been talking to could turn out to be an appropriate action towards this inarguable important direction.

REFERENCES

[1] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. 2007. Deciding Equivalences Among Conjunctive Aggregate Queries. *J. ACM* (2007).

[2] Corinna Cortes, L. D. Jackel, Sara A. Solla, Vladimir Vapnik, and John S. Denker. 1993. Learning curves: asymptotic values and rate of convergence. (1993). http://dl.acm.org/citation.cfm?id=2987189.2987231

[3] David Haussler, Michael Kearns, H. Sebastian Seung, and Naftali Tishby. 1996. Rigorous Learning Curve Bounds from Statistical Mechanics. *Machine Learning* 25, 2/3 (1996), 195–236. DOI: http://dx.doi.org/10.1023/A:1026499208981

[4] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014).

[5] Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2015. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD*.

[6] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. MLog: Towards Declarative In-Database Machine Learning. *ArXiv* (2017).

[7] Kevin Schawinski, Ce Zhang, Hantian Zhang, Lucas Fowler, and Gokula Krishnan Santhanam. 2017. Generative adversarial networks recover features in astrophysical images of galaxies beyond the deconvolution limit. *Monthly Notices of the Royal Astronomical Society: Letters* (2017).