**ModelHub: Lifecycle Management for Deep Learning**

Hui Miao, Ang Li, Larry S. Davis, Amol Deshpande
University of Maryland, College Park, MD, USA
\{hui, angli, lsd, amol\}@cs.umd.edu

**ABSTRACT**

Deep learning has improved state-of-the-art results in many important fields, and has been the subject of much research in recent years, leading to the development of several systems for facilitating deep learning. Current systems, however, mainly focus on model building and training phases, while the issues of data management, model sharing, and lifecycle management are largely ignored. Deep learning modeling lifecycle contains a rich set of artifacts, such as learned parameters and training logs, and frequently conducted tasks, e.g., to understand the model behaviors and to try out new models. Dealing with such artifacts and tasks is cumbersome and left to the users. To address these issues in a comprehensive manner, we propose ModelHub, which includes a novel model versioning system (DLV); a domain specific language for searching through model space (DQL); and a hosted service (ModelHub) to store developed models, explore existing models, enumerate new models and share models with others.

1. INTRODUCTION

Deep learning models (also called **deep neural networks**) have dramatically improved state-of-the-art results for many important reasoning and learning tasks including speech recognition, object recognition, and natural language processing in recent years [10]. Learned using massive amounts of training data, DNN models have superior generalization capabilities, and the intermediate layers in many deep learning models have been proven useful in providing effective semantic features that can be used with other learning techniques. However, there are many critical large-scale data management issues in learning, storing, sharing, and using deep learning models, which are largely ignored by researchers today, but are coming to the forefront with the increased use of deep learning in a variety of domains. In this proposal, we discuss some of those challenges, and propose a comprehensive platform to address them. Given the large scale of data involved (both training data and the learned models themselves), we argue that database researchers should play a much larger role in this area.

**Deep Neural Networks:** We begin with a brief, simplified overview. A deep learning model is a deep neural network (DNN) consisting of many layers having nonlinear activation functions that are capable of representing complex transformations between input data and desired output. Let \( \mathbb{D} \) denote a data domain and \( \mathbb{O} \) denote a prediction label domain (e.g., \( \mathbb{D} \) may be a set of images; \( \mathbb{O} \) may be the names of the set of objects we wish to recognize). As with any prediction model, a DNN is a mapping function \( f : \mathbb{D} \rightarrow \mathbb{O} \) that minimizes a certain loss function \( L \), and is of the following form:

\[
\begin{align*}
    f_0 &= \sigma_0(W_0d + b_0) \\
    f_i &= \sigma_i(W_if_{i-1} + b_i) \quad 0 < i \leq n \\
    L(f_n, l_d) &= \text{loss function} \\
\end{align*}
\]

Here \( i \) denotes the layer number, \((W_i, b_i)\) are learnable weights and bias parameters in layer \( i \), and \( \sigma_i \) is an activation function that non-linearly transforms the result of the previous layer (common activation functions include sigmoid, ReLU, etc.). Given a learned model and an object \( d \), applying \( f_0, f_1, \ldots, f_n \) in order gives us the prediction label for that object. In the training phase, the model parameters are learned by minimizing \( L(f_n, l_d) \), typically done through iterative methods, such as **stochastic gradient descent**.

**DNN Modeling Lifecycle and Challenges:** Compared with the traditional approach of feature engineering followed by model learning [14], deep learning is an end-to-end learning approach, i.e., the features are not given by a human but learned in an automatic manner from the input data. Moreover, the features are complex and have a hierarchy along with the network representation. This requires less domain expertise and experience from the modeler, but understanding and explaining the learned models is difficult; why even well-studied models work so well is still a mystery and under active research. Thus, when developing new models, changing the learned model (especially its network structure and hyper-parameters) becomes an empirical search task.

In Fig. 1, we show a typical deep learning modeling lifecycle. Given a prediction task, a modeler often starts from well-known models which have been successful in similar task domains; she then specifies input training data and output loss functions, and repeatedly adjusts the DNN on operators and connections like Lego bricks, tunes model hyper-parameters, trains and evaluates the model, and repeats this loop until prediction accuracy does not improve. Due to a lack of understanding about why models work, the adjustments and tuning inside the loop are driven by heuristics, e.g., adjusting hyper-parameters that appear to have a significant impact on the learned weights, applying novel layers or tricks seen in recent empirical studies, and so on. Thus, many similar models are trained and compared, and a series of model variants need to be explored and developed. Due to the expensive learning/training phase, each iteration of the modeling loop takes a long period of time and produces many (checkpointed) snapshots of the model.

The modeling lifecycle exposes several systems and data management challenges, which include:

- It is difficult to keep track of the many models developed and/or understand the differences amongst them. It is common to see a modeler write all configurations in a spreadsheet to keep track of temporary folders of input, setup scripts, snapshots and logs,
- The development lifecycle itself has time-consuming repetitive sub-steps, such as adding a layer at different places to adjust...
We propose the ModelHub system to address these challenges. It consists of three key components: (a) a model versioning system (DLV) to store and query the models and their versions, (b) a model enumeration and hyper-parameter tuning domain specific language (DQL) to serve as an abstraction layer to help modelers focus on the creation of the models instead of repetitive steps in the lifecycle, (c) a hosted deep learning model sharing system (ModelHub) to publish, discover and reuse models from others. The ModelHub system is designed to be used with current training systems (e.g., caffe) for both learning and evaluation.

We focus on describing the key ModelHub functionality in this proposal. In ongoing work, we are developing novel storage techniques for compactly storing large numbers of such models, algorithms for aligning two similar models, and approaches for efficiently executing complex DQL queries and searching through the user-specified search space.

Related Work: There have been several high-profile deep learning systems in recent years, but those typically focus on training aspects (e.g., distributed training, utilizing GPUs, etc.) [8, 6, 1, 4]. The data management and lifecycle management challenges discussed above have been largely ignored so far, but are becoming critical as the use of deep learning permeates through a variety of application domains, since those pose a high barrier to entry for many potential users. In the database community, there has been increasing work on developing general-purpose systems for supporting machine learning [9, 12, 11], including pushing predictive models into databases [2, 7], accelerating learning by optimizing physical design [14], and managing modeling lifecycles and serving predictive models in advanced ways [13, 5]. ModelHub is motivated by similar principles; aside from a specific focus on deep learning models, ModelHub also supports versioning as a first-class construct [3] which differentiates it from that work.

2. MODELHUB OVERVIEW

### System Architecture

We show the ModelHub architecture including the key components and their interactions in Fig. 2. DLV is a version control system (VCS) implemented as a command-line tool (dlv), that serves as an interface to interact with the rest of the components. Use of a specialized VCS such as git or svn allows us to better portray and query the internal structure of the artifacts generated in a modeling lifecycle, such as network definitions, training logs, binary weights, and relationships between models. The key utilities of dlv are listed in Table 2.1, grouped by their purpose; we explain these in further detail in Sec. 2.3. DQL is a DSL we propose to assist modelers in deriving new models; the DQL parser and optimizer components in the figure are used to support this language. The model learning module interacts with external deep learning tools that the modeler uses for training and testing. In the demonstration, we implement a concrete model learning module on top of caffe, which is a popular deep learning training system for computer vision models [8]. Finally, the ModelHub service is a hosted toolkit to support publishing, discovering and reusing models, and serves similar role for DNN models as github for software development or DataHub for data science [3].

### Data Model

ModelHub works on two levels of data models: conceptual DNN model, and data model for the model versions in the DLV repository.

**DNN Model:** A DNN model can be understood in different ways, as one can tell from the different model creation APIs in popular deep learning systems. In the formulation mentioned in Sec. 1, if we view a function \( f \) as a node and dependency relationship \( (f_i, f_o) \) as an edge, it becomes a directed acyclic graph (DAG). Depending on the granularity of the function in the DAG, either at the tensor arithmetic operator level (add, multiply), or at a logical composition of those operators (convolution layer, full layer), it forms different types of DAGs. In ModelHub, we consider a DNN model node as a composition of unit operators (layers), often adopted by computer vision models. The main reason for the decision is that we focus on the productivity improvement in the lifecycle, rather than the implementation efficiencies for training and testing.

**VCS Data Model:** When managing DNN models in the VCS repository, a model version represents the contents in a single version. It consists of a network definition, a collection of weights (each of which is a value assignment for the weight parameters), a set of extracted metadata (such as hyper-parameter, accuracy and loss generated in the training phase), and a collections of files used together with the model instance (e.g., scripts, datasets). In addition, we enforce that a model version must be associated with a human readable name for better utility, which reflects the logical groups of a series of improvement efforts over a DNN model in practice.
In the implementation, model versions can be viewed as a relation M(name, id, N, W, M, F), where id is part of the primary key of model versions and is auto-generated to distinguish model versions with the same name. In brief, N, W, M, F are the network definition, weight values, extracted metadata and associated files respectively. The DAG, N, is stored with two EDBs Node(id, node, A), where A is a list of attributes such as name, and Edge(from, to), W, M, F is not discussed in detail in the demonstration proposal due to space limits. Besides a set of model versions, the lineage of the model versions are captured using a separate parent relation, P. All of these relations need to be maintained/updated when the user runs the different dlv commands that update the repository.

2.3 Query Facilities

The query facilities we provide can be categorized into two types: a) model exploration queries and b) model enumeration queries.

2.3.1 Model Exploration Queries

Model exploration queries interact with the models in a repository, and the users use them to understand a particular model, query lineages of the models, and compare several models. For usability, we design it as query templates via dlv sub-command with options, similar to other VCS. We describe the queries in the following sub-sections followed by query templates with most important options.

List Models & Related Lineages: By default, the query lists all versions of all models including their commit descriptions and parent versions; it also takes options, such as showing results for a particular model, or limiting the number of versions to be listed.

```bash
dlv list [-model_name] [-commit_msg] [--last]
```

Describe Model: dlv desc shows the modeler extracted metadata from a model version, such as the network definition, learnable parameters, execution footprint (memory and runtime), activations of convolution networks, weight matrices, and evaluation results across iterations. Note the activation is the intermediate output of a DNN model in computer vision and often used as an important tool to understand the model. The current output formats are a result of discussions with computer vision modelers to deliver tools that fit their needs. In addition to printing to console, the query supports HTML output for displaying the images and visualizing the weight distribution, part of which is shown in Fig. 3.

```bash
dlv desc [-model_name] [-output]
```

Compare Models: dlv diff takes a list of model names or version ids and allows the modeler to compare the DNN models. Most of descri components are aligned and returned in the query result side by side. One challenge as well as motivation of model comparison, is that the models often have subtle differences, and an alignment needs to be done before composing the result.

```bash
dlv diff [-model_names] [-versions] [-output]
```

Evaluate Model: dlv eval runs test phase of the managed models with an optional config specifying different data or changes in the current hyper-parameters. The main usages of exploration query are two-fold: 1) for the users to get familiar with a new model, 2) for the user to test known models on different data or settings. The query returns the accuracy and optionally the activations. It is worth pointing out that complex evaluations can be done via DQL.

```bash
dlv eval [-model_name] [-versions] [-config]
```

2.3.2 Model Enumeration Queries

Model enumeration queries are used to explore variations of currently available models in a repository by changing network structures or tuning hyper-parameters. There are several operations that need to be done in order to derive new models: 1) Select models from the repository to improve; 2) Slice particular models to get reusable components; 3) Construct new models by mutating the existing ones; 4) Try the new models on different hyper-parameters and pick good ones to save and work with. When enumerating models, we also want to stop exploration of bad models early.

To support this rich set of requirements, we propose the DQL domain-specific language, that can be executed using “dlv query”. Challenges of designing the language are a) the data model is mixed with relational and the graph data models and b) the enumeration include hyper-parameter tuning as well as network structure mutations, which are very different operations. A thorough explanation of the language model is beyond the scope of the demo proposal, instead we show the key operators and constructs along with a set of examples (Query 1–4) to show how requirements are met.

**Query 1:** DQL select query to pick the models.

```sql
select m
where m.name like "alexnet_\%" and m.creation_time > "2015-11-22" and m["conv1,3,5"]
```

**Query 2:** DQL slice query to get a sub-network.

```sql
slice m2 from m
where m.name like "alexnet-origin\%"
mutate m2.input = m1["conv1"] and m2.output = m1["fc7"]
```

**Query 3:** DQL construct query to derive more models on existing ones.

```sql
construct m2 from m1
where m1.name like "alexnet-avgv1\%" and m1["conv*(\$1)"].next has POOL("MAX")
mutate m1["conv*(\$1)"].insert = RELU("relu\$1")
```

**Query 4:** DQL evaluate query to enumerate models with different network definitions, search hyper-parameters, and eliminate models.

```sql
evaluate m
from "query3"
with config = "path to config"
where config.base_lr in [0.1, 0.01, 0.001] and config.net["conv*"].lr auto and config.input_data in ["path1", "path2"]
keep top(5, m["loss"], 100)
```

**Key Operators:** We adopt the standard SQL syntax to interact with the repository. DQL views the repository as a single model version table. A model version instance is a DAG, which can be viewed as object types in modern SQL conventions. In DQL, attributes can be referenced using attribute names (e.g. m1.name, m1.creation_time, m2.input, m2.output), while navigating the internal structures of the DAG, i.e. the Node and Edge EDB, we provide a regexp style `select` operator on a model version to access individual DNN nodes, e.g. m1("conv[1,3,5]") in Query 1 filters the nodes in m1. Once the selector operator returns a set of nodes, prev and next attributes of the node allow 1-hop traversal in the DAG. Note that POOL("MAX") is one of the standard built-in node templates for condition clauses. Using SPJ operators with object type `attribute access` and the `selector operator`, we allow relational queries to be mixed with graph traversal conditions.

To retrieve reusable components in a DAG, and mutate it to get new models, we provide `slice`, `construct` and `mutate` operators. Slice originates in programming analysis research; given an input and an output node, it returns a subgraph including all paths from the input to the output and the connections which are needed to produce the output. Construct can be found in graph query languages such as SPARQL to create new graphs. We allow `construct` to derive new DAGs by using selected nodes to `insert` nodes inside an outgoing edges or to `delete` outgoing edge connecting to another node.
shown, while on the right side, the activation responses and the weight matrix of each channel are shown.

After using `dlv` to manage and understand the models, our demonstration includes enumerating new models using DQL by running the queries like the ones listed in Sec. 2. On the other hand, without DQL, the default approach is illustrated for experienced users, such as editing the model, setting up running instances and parsing results, and how the DQL does related steps behind the scenes.

Finally, we show how to publish and download models from ModelHub. Modified face recognition models are published via `dlv publish`. Besides the face dataset, we also prepare well-known models for other popular image datasets, such as MNIST, and host them beforehand in a ModelHub instance. The attendees are allowed to discover and download models via `dlv search` and `pull`.

As the take-away message, we hope the attendees can experience the development of deep learning models for real-world applications, get an idea about the required skill sets in model development and the awkwardness of the current systems w.r.t. the lifecycle management, observe the subtle differences in the models created, and understand the importance of the problems we propose and the need for more principled data management approaches for addressing those. We also hope to derive additional features from the feedback of potential users among the attendees.

4. REFERENCES


