2017 IEEE 33rd International Conference on Data Engineering

Towards Unified Data and Lifecycle Management for Deep Learning

Hui Miao, Ang Li, Larry S. Davis, Amol Deshpande
Department of Computer Science
University of Maryland, College Park, MD 20742
{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 generates a rich set of data artifacts, e.g., learned parameters and training logs, and it comprises of several 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 largely left to the users. This paper describes our vision and implementation of a data and lifecycle management system for deep learning. First, we generalize model exploration and model enumeration queries from commonly conducted tasks by deep learning modelers, and propose a *high-level domain specific language (DSL)*, inspired by SQL, to raise the abstraction level and thereby accelerate the modeling process. To manage the variety of data artifacts, especially the large amount of checkpointed float parameters, we design a novel model versioning system (`dlv`), and a read-optimized parameter archival storage system (PAS) that minimizes storage footprint and accelerates query workloads with minimal loss of accuracy. PAS archives versioned models using deltas in a multi-resolution fashion by separately storing the less significant bits, and features a novel progressive query (inference) evaluation algorithm. Third, we develop efficient algorithms for archiving versioned models using deltas under *co-retrieval constraints*. We conduct extensive experiments over several real datasets from computer vision domain to show the efficiency of the proposed techniques.**

I. Introduction

Deep learning models, also called *deep neural networks* (DNN), have dramatically improved the state-of-the-art results for many important reasoning and learning tasks including speech recognition, object recognition, and natural language processing in recent years [1]. 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 and are applicable to other problems. 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 paper, we discuss some of those challenges in the context of the modeling lifecycle, and propose a comprehensive system to address them. Given the large scale of data involved (both training data and the learned models themselves) and the increasing need for high-level declarative abstractions, we argue that database researchers should play a much larger role in this area. Although this paper primarily focuses on deep neural networks, similar data management challenges are seen in lifecycle management of others types of ML models like logistic regression, matrix factorization, etc.

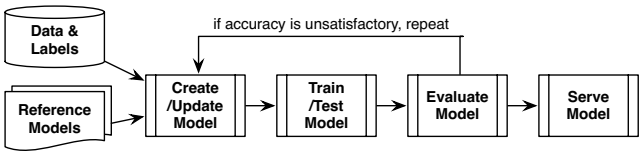


Fig. 1. Deep Learning Modeling Lifecycle

DNN Modeling Lifecycle and Challenges: Compared with the traditional approach of *feature engineering* followed by *model training* [2], deep learning is an end-to-end learning approach, i.e., the features are not given by a human but are 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 *hyperparameters*) becomes an empirical search task.

In Fig. 1, we show a typical deep learning modeling lifecycle (we present an overview of deep neural networks in the next section). Given a prediction task, a modeler often starts from well-known models that 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 hyperparameters, 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 hyperparameters 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 needs 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. As noted above, this is a common workflow across many other ML models as well.

Current systems (Caffe [3], Theano, Torch, TensorFlow [4], etc.) mainly focus on model building and training phases, while

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.112

the issues of data management, model sharing, and lifecycle management are largely ignored. Modelers are required to write external imperative scripts, edit configurations by hand and manually maintain a manifest of model variations that have been tried out; not only are these tasks irrelevant to the modeling objective, but they are also challenging and nontrivial due to the complexity of the model as well as large footprints of the learned models. More specifically, the tasks and data artifacts in the modeling lifecycle expose several systems and data management challenges, which include:

- **Understanding and Comparing Models**: It is difficult to keep track of the many models developed and/or understand the differences amongst them. Differences among both the metadata about the model (training sample, hyperparameters, network structure, etc.), as well as the actual learned parameters, are of interest. 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, which is not only a cumbersome but also an error-prone process.
- **Repetitive Adjusting of Models**: The development lifecycle itself has time-consuming repetitive sub-steps, such as adding a layer at different places to adjust a model, searching through a set of hyperparameters for the different variations, reusing learned weights to train models, etc., which currently have to be performed manually.
- **Model Versioning**: Similar models are possibly trained and run multiple times, reusing others' weights as initialization (finetuning) [5], [24]. Maintaining the different model versions generated over time and their relationships can help with identifying errors and concept drifts, comparing models over new inputs, and potentially reverting back to a previous model. Even for a single learned model, storing the different checkpointed snapshots can help with "warm-start" and can provide important insights into the training processes.
- **Parameter Archiving**: The storage footprint of deep learning models tends to be very large. Recent top-ranked models in the ImageNet task have billions of floating-point parameters and require hundreds of MBs to store one snapshot during training. Due to resource constraints, the modeler has to limit the number of snapshots, even drop all snapshots of a model at the cost of retraining when needed.

In addition, although not a focus of this paper, sharing and reusing models is not easy, especially because of the large model sizes and specialized tools used for learning.

ModelHub: In this paper, we describe our ModelHub system that attempts to address these challenges in a holistic fashion. ModelHub is not meant to replace popular training-focused DNN systems, but rather designed to be used with them to accelerate modeling tasks and manage the rich set of lifecycle artifacts. It consists of three key components: (a) a model versioning system (`DLV`) to store, query and aid in understanding the models and their versions, (b) a model network adjustment and hyperparameter tuning domain specific language (`DQL`) to serve as an abstraction layer to help modelers focus on the creation of the models, (c) a hosted deep learning model sharing system (ModelHub) to exchange `DLV` repositories and enable publishing, discovering and reusing models from others.

Some of the key features and innovative design highlights of ModelHub are as follows. **(a)** We propose a `git`-like VCS interface, familiar to most, to let the modeler manage and explore the created models in a repository, and an `SQL`-like model enumeration DSL to aid modelers in making and examining multiple model adjustments easily. **(b)** Behind the declarative constructs, ModelHub manages different artifacts in a split back-end storage. Structured data, such as network structure, training logs of a model, lineages of different model versions, output results, are stored in *a relational database*; while learned float-point parameters of a model are viewed as a set of float matrices and managed in *a read-optimized archival storage* (PAS). **(c)** Parameters dominate the storage footprint and floats are well-known at being difficult to compress. We study PAS implementation thoroughly under the context of DNN query workload and advocate a segmented approach to store the learned parameters, where the low-order bytes are stored independently of the high-order bytes. We also develop novel model evaluation schemes to use high order bytes solely and progressively uncompress less-significant chunks if needed to ensure the correctness of an inference query. **(d)** Archiving versioned models using *deltas* exhibits a new type of dataset versioning problem which not only features the familiar storage-access trade-off but also model-level constraints, which we capture and optimize for. **(e)** Finally, the VCS model repository design extends naturally to a collaborative format and to an online system that contains rich model lineages and enables sharing, reusing, reproducing DNN models.

ModelHub is an end-to-end system where we introduce new high-level abstractions suitable for automating deep learning workflows, and show how those abstractions can be implemented efficiently to manage and optimize the different steps in the workflow. We note that our high-level abstractions could be supported over a different backend (e.g., where all the data artifacts are simply written to a file system); similarly, our storage backend, and the optimizations therein, are independent of those abstractions to a large degree and could be useful for any collection of multi-dimensional arrays (*tensors*). At the same time, thinking about the two components simultaneously allows us to better inform the design decisions that can be made, and opens up new optimization opportunities.

Contributions: Our key research contributions are:

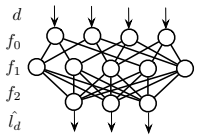
- We propose the first comprehensive DNN lifecycle management system, study its design requirements, and propose declarative constructs (`DLV` and `DQL`) to provide high-level abstractions.
- We develop PAS, a read-optimized archival storage system for dealing with a large collection of versioned float matrices.
- We formulate a new dataset versioning problem with *co-usage constraints*, analyze its complexity, and design efficient algorithms for solving it.
- We develop a progressive, approximate query evaluation scheme that avoids reading low-order bytes of the parameter matrices unless necessary.
- We present a comprehensive evaluation of ModelHub that shows the proposed techniques are useful for real-life models, and scale well on synthetic models.

Outline: In Section II, we provide background on related topics in DNN modeling lifecycle. In Section III, we present an overview of ModelHub, and discuss the declarative interfaces. We describe the parameter archival store (PAS) in Section IV, present an experimental evaluation in Section V, and closely related work in Section VI.

II. Background

To support our design decisions, we overview the artifacts and common task practices in DNN modeling lifecycle.

Deep Neural Networks: A deep learning model, or a deep neural network (DNN), consists 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, i.e, *labels*). As with any prediction model, a DNN is a mapping function $f : \mathbb{D} \rightarrow \mathbb{O}$ that minimizes a loss function L, and is of the following form:



$$f_0 = \sigma_0(W_0 d + b_0) \qquad d \in \mathbb{D}$$
$$f_i = \sigma_i(W_i f_{i-1} + b_i) \qquad 0 < i \leq n$$
$$L(f_n, l_d) \qquad l_d \in \mathbb{O}$$

Here i denotes the layer number, (W_i, b_i) are learnable weights and bias parameters in layer i, and σ_i is an activation function that non-linearly transforms the result of the previous layer (common activation functions include sigmoid, ReLU, etc.). More formally, a layer, $L_i : (W, H, X) \mapsto Y$, is a function which defines data transformations from *tensor* X to tensor Y. W are the parameters which are learned from the data, and H are the hyperparameters which are given beforehand. A layer is non-parametric if $W = \emptyset$.

Given a learned model and an input d, applying $f_0, f_1, ..., f_n$ in order gives us the prediction label for that input data. 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*.

Fig. 2 shows a classic *convolutional DNN*, LeNet, used to assign digit labels, $\{0 \cdots 9\}$, to handwritten images. A cube represents an intermediate tensor, while the dotted lines are unit transformations between tensors. In the computer vision community, the layers defining the transformations are considered building blocks of a DNN model, and are referred to using conventional names such as *full* layer, *convolution* layer, *pool* layer, *normalization* layer, etc. The chain is often called the network *architecture*. The LeNet architecture has two convolution layers, each followed by a pool layer, and two full layers. Moreover, winning models in recent ILSVRC (ImageNet Large Scale Vision Recognition Competitions) are shown in Table I, with their architectures described by a composition of common layers in regular expressions syntax for illustrating the similarities (the activation functions and detailed connections are omitted).

DNN models are learned from massive data based on some architecture, and modern successful computer vision DNN architectures consist of a large number of float weight parameters (*flops*) shown in Table I, resulting in large storage footprints (GBs) and long training times (often weeks). Furthermore, the training process is often checkpointed and variations of models need to be explored, leading to many model copies.

Modeling Data Artifacts: DNN modeling results in a mixture of data artifacts, including structured datasets, float matrices, and script files. Some key data artifacts include: (a) the values of the *hyperparameters* (e.g., learning rate, momentum) used by the optimization algorithm; (b) *learning measurements* (e.g., per-iteration objective loss values, accuracy scores) collected in various logs; (c) *trained snapshots* which are typically not deleted given the expensive training phase and may be reused as initializations for training or *fine-tuning* later; and (d) *arbitrary files* including hand-crafted scripts, result spreadsheets, and other types of provenance information that may be needed for exploring and analyzing models, and for reproducing results [6].

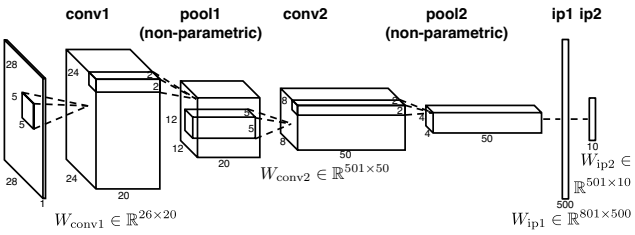


Fig. 2. Anatomy of A DNN Model (LeNet)

Network	**Architecture** (in regular expression)	$\|W\|$ **(flops)**
LeNet [7]	$(L_{conv}L_{pool})\{2\}L_{ip}\{2\}$	4.31×10^5
AlexNet [8]	$(L_{conv}L_{pool})\{2\}(L_{conv}\{2\}L_{pool})\{2\}L_{ip}\{3\}$	6×10^7
VGG [9]	$(L_{conv}\{2\}L_{pool})\{2\}(L_{conv}\{4\}L_{pool})\{3\}L_{ip}\{3\}$	1.96×10^{10}
ResNet [10]	$(L_{conv}L_{pool})(L_{conv})\{150\}L_{pool}L_{ip}$	1.13×10^{10}

TABLE I. Popular CNN Models for Object Recognition

Model Adjustment: In a modeling lifecycle for a prediction task, the *update-train-evaluate* loop is repeated in daily work, and many model variations are adjusted and trained. In general, once data and loss are determined, model adjustment can be done in two orthogonal steps: a) network architecture adjustments where layers are dropped or added and layer function templates are varied, and b) hyperparameter selections, which affect the behavior of the optimization algorithms. There is much work on search strategies to enumerate and explore both.

Model Sharing: Due to good generalizability, long training times, and verbose hyperparameters required for large DNN models, there is a need to share the trained models. Jia et al. [3] built an online venue (Caffe Model Zoo) to share models. Briefly, Model Zoo is part of a GitHub repository[1] with a markdown file edited collaboratively. To publish models, modelers add an entry with links to download trained parameters in `caffe` format. Apart from the caffe community, similar initiatives are in place for other training systems.

III. ModelHub System Overview

We show the ModelHub architecture including the key components and their interactions in Fig. 3. At a high level, the ModelHub functionality is divided among a local component and a remote component. The local functionality includes the integration with popular DNN systems such as `caffe`, `torch`, `tensorflow`, etc., on a local machine or a cluster. The remote functionality includes sharing of models, and their versions, among different groups of users. We primarily focus on the local functionality in this paper.

[1]Caffe Model Zoo: https://github.com/BVLC/caffe/wiki/Model-Zoo

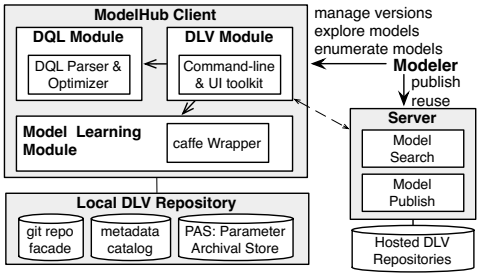


Fig. 3. ModelHub System Architecture

On the local system side, 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 local and remote components. Use of a specialized VCS instead of a general-purpose 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 III-A, grouped by their purpose; we explain these in further detail in Sec. III-B. DQL is a DSL we propose to assist modelers in deriving new models; the DQL query 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. They are essentially wrappers on specific DNN systems that extract and reproduce modeling artifacts. 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 [11].

A. Data Model

ModelHub works with two data models: a conceptual DNN model, and a data model for the versions in a 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. I, if we view a function f_i as a node and dependency relationship (f_i, f_{i-1}) 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 this decision is that we focus on productivity improvement in the lifecycle, rather than implementation efficiencies of 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 hyperparameter, accuracy and loss generated in the training phase), and a collection 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.

Type	Command	Description
model version management	`init`	Initialize a dlv repository.
	`add`	Add model files to be committed.
	`commit`	Commit the added files.
	`copy`	Scaffold model from an old one.
	`archive`	Archive models in the repository.
model exploration	`list`	List models and related lineages.
	`desc`	Describe a particular model.
	`diff`	Compare multiple models.
	`eval`	Evaluate a model with given data.
model enumeration	`query`	Run DQL clause.
remote interaction	`publish`	Publish a model to ModelHub.
	`search`	Search models in ModelHub.
	`pull`	Download from ModelHub.

TABLE II. A list of key dlv utilities.

In the implementation, model versions can be viewed as a relation *model_version*(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 as two tables: *Node*(id, node, A), where A is a list of attributes such as layer name, and *Edge*(from, to). W is managed in our learned parameter storage (PAS, Sec. IV). M, the metadata, captures the provenance information of training and testing a particular model; it is extracted from training logs by the wrapper module, and includes the hyperparameters when training a model, the loss and accuracy measures at some iterations, as well as dynamic parameters in the optimization process, such as learning rate at some iterations. Finally, F is file list marked to be associated with a model version, including data files, scripts, initial configurations, and etc. Besides a set of *model versions*, the lineage of the *model versions* are captured using a separate *parent*(base, derived, commit) relation. All of these relations are maintained/updated in a relational backend when the modeler runs the different dlv commands that update the repository. Managing the provenance and metadata information, and supporting analysis queries over that, is a challenging problem in itself [6].

B. Query Facilities

Once the DNN models and their relationships are managed in DLV, the modeler can interact with them easily. The query facilities we provide can be categorized into two types: a) model exploration queries and b) model enumeration queries.

1) Model Exploration Queries: Model exploration queries interact with the models in a repository, and are used to understand a particular model, to query lineages of the models, and to compare several models. For usability, we design it as query templates via dlv sub-command, similar to other VCS. In addition, we render results in HTML front end when needed. For example, `dlv list` finds models and lineages; `dlv desc` queries the extracted metadata from a model version, such as the network definition, learnable parameters, execution footprint and evaluation results across iterations; `dlv diff` to compare models side by side via the metadata; and `dlv eval` to run test phase of the managed models with given data points. We refer the reader to [12] for more details.

```
select m1
where  m1.name like "alexnet_%" and
       m1.creation_time > "2015-11-22" and
       m1["conv[1,3,5]"].next has POOL("MAX")
```

Query 1. DQL select query to pick the models.

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

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

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

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

```
evaluate m
from "query3"
with config = "path to config"
vary 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)
```

Query 4. DQL evaluate query to enumerate models with different network architectures, search hyperparameters, and eliminate models.

2) Model Enumeration Queries: Model enumeration queries, specified using our DQL domain specific language, are used to explore variations of currently available models in a repository by changing the network structures or tuning hyperparameters. DQL queries are executed using "`dlv query`". At a high level, there are four key operations that need to be performed, which we use to design DQL: (1) *Select* models from the repository by filtering on metadata and/or other properties like accuracy; (2) *Slice* one or more models to get reusable components; (3) *Construct* new models by modifying and stitching together those components; and (4) *Try* the new network architectures on different sets of hyperparameters and pick good ones to save and work with. When enumerating models, we also want to stop exploration of bad models early. However, designing a query language that satisfies this rich set of requirements is challenging because: (a) the data model is a mix of relational and graph data models and (b) the enumeration includes hyperparameter tuning as well as network structure mutations, which are very different operations. We omit a thorough explanation of the language due to space constraints, and instead show the key operators and constructs along with a set of examples (Query 1~4) to show how requirements are met.

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 *selector 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 a start and an end node, it returns a subgraph including all paths from the start to the end 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 by splitting an outgoing edge or to *delete* an outgoing edge connecting to another node. **Mutate** limits the places where *insert* and *delete* can occur. For example, Query 2 and 3 generate reusable subgraphs and new graphs. Query 2 slices a sub-network from matching models between convolution layer 'conv1' and full layer 'fc7', while Query 3 derives new models by appending a ReLU layer after all convolution layers followed by an average pool. All queries can be nested.

Finally, **evaluate** can be used to try out new models, with potential for early out if expectations are not reached. We separate the network enumeration component from the hyperparameter turning component; while network enumeration can be nested in the *from* clause, we introduce a *with operator* to take an instance of a tuning config template, and a *vary operator* to express the combination of activated multi-dimensional hyperparameters and search strategies. *auto* is keyword implemented using default search strategies (currently grid search). To stop early and let the user control the stopping logic, we introduce a *keep operator* to take a rule consisting of stopping condition templates, such as top-k of the evaluated models, or accuracy threshold. Query 4 evaluates the models constructed and tries combinations of at least three different hyperparameters, and keeps the top 5 models w.r.t. the loss after 100 iterations.

C. ModelHub Implementation

On the local side, the current implementation of ModelHub maintains the data in multiple back-ends and utilizes `git` to manage the non-parameter files. Queries are decomposed and sent to different backends and chained accordingly. As the model repository is standalone, we host the repositories as a whole in a ModelHub service. The modeler can use the `dlv publish` to push the repository for archiving, collaborating or sharing, and use `dlv search` and `dlv pull` to discover and reuse remote models. We envision such a form of collaboration can facilitate a learning environment, as all versions in the lifecycle are accessible and understandable with ease.

IV. Parameter archival storage (PAS)

Modeling lifecycle for DNNs, and machine learning models in general, is centered around the learned parameters, whose storage footprint can be very large. The goal of PAS is to maintain a large number of learned models as compactly as possible, without compromising the query performance. Before introducing our design, we first discuss the queries of interest, and some key properties of the model artifacts (IV-A). We then describe different options to store a single *float matrix*, and to construct *deltas (differences)* between two matrices

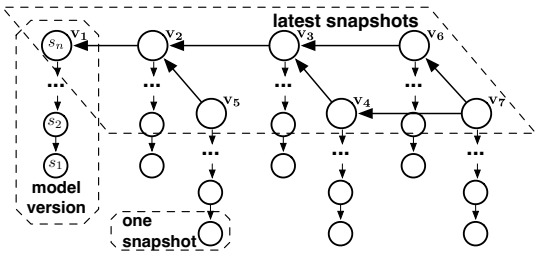


Fig. 4. Relationship between Model Versions and Snapshots

(IV-B). We then formulate the optimal version graph storage problem, discuss how it differs from the prior work, and present algorithms for solving it (IV-C). Finally, we develop a novel approximate model evaluation technique, suitable for the segmented storage technique that PAS uses (IV-D).

A. Weight Parameters & Query Type of Interests

We illustrate the key weight parameter artifacts and the relationships among them in Fig. 4, and also explain some of the notations used in this section. At a high level, the predecessor-successor relationships between all the developed models is captured as a **version graph**. These relationships are user-specified and conceptual in nature, and the interpretation is left to the user (i.e., an edge $v_i \rightarrow v_j$ indicates that v_j was an updated version of the model that the user checked in after v_i, but the nature of this update is irrelevant for storage purposes). A model version v_i itself consists of a series of snapshots, $s_1, ..., s_n$, which represent checkpoints during the training process (most systems will take such snapshots due to the long running times of the iterations). We refer the last or the best checkpointed snapshot s_n as the **latest snapshot** of v_i, and denote it by s_{v_i}.

One snapshot, in turn, consists of intermediate data X and trained parameters W (e.g., in Fig. 2, the model has 431080 parameters for W, and $19694 \cdot b$ dimensions for X, where b is the minibatch size). Since X is useful only if training needs to be resumed, only W is stored in PAS. Outside of a few rare exceptions, W can always be viewed as a collection of float matrices, $\mathbb{R}^{m \times n}, m \geq 1, n \geq 1$, which encode the weights on the edges from outputs of the neurons in one layer to the inputs of the neurons in the next layer. Thus, we treat a *float matrix* as a first class data type in PAS[2].

The retrieval queries of interest are dictated by the operations that are done on these stored models, which include: (a) testing a model, (b) reusing weights to fine-tune other models, (c) comparing parameters of different models, (d) comparing the results of different models on a dataset, and (e) model exploration queries (Sec. III-B). Most of these operations require execution of **group retrieval** queries, where all the weight matrices in a specific snapshot need to be retrieved. This is different from range queries seen in array databases (e.g., SciDB), and also have unique characteristics that influence the storage and retrieval algorithms:

- *Similarity among Fine-tuned Models*: Although non-convexity of the training algorithm and differences in network architectures across models lead to non-correlated parameters, the widely-used fine-tuning practices (Sec. II) generate model versions with similar parameters, resulting in efficient delta encoding schemes.
- *Co-usage constraints:* Prior work on versioning and retrieval [13] has focused on retrieving a single artifact stored in its entirety. However, we would like to store the different matrices in a snapshot independently of each other, but we must retrieve them together. These co-usage constraints make the prior algorithms inapplicable as we discuss later.
- *Low Precision Tolerance*: DNNs are well-known for their tolerance to using low-precision floating point numbers (Sec. VI), both during training and evaluation. Further, many types of queries (e.g., visualization and comparisons) do not require retrieving the full-precision weights.
- *Unbalanced Access Frequencies*: Not all snapshots are used frequently. The latest snapshots with the best testing accuracy are used in most of the cases. The checkpointed snapshots have limited usages, including debugging and comparisons.

B. Parameters As Segmented Float Matrices

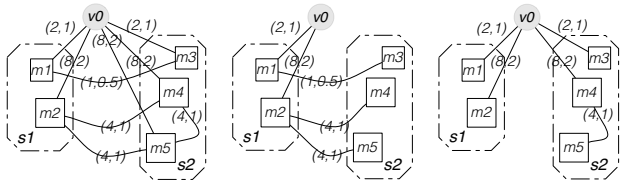
Float Data Type Schemes: Although binary (1/-1) or ternary (1/0/-1) matrices are sometimes used in DNNs, in general PAS handles real number weights. Due to different usages of snapshots, PAS offers a handful of float representations to let the user trade-off storage efficiency with lossyness using `dlv`.

- *Float Point*: DNNs are typically trained with single precision (32 bit) floats. This scheme uses the standard IEEE 754 floating point encoding to store the weights with sign, exponent, and mantissa bits. IEEE half-precision proposal (16 bits) and tensorflow truncated 16bits [4] are supported as well and can be used if desired.
- *Fixed Point*: Fixed point encoding has a global exponent per matrix, and each float number uses k bits to represent sign and mantissa. This scheme is lossy as tail positions are dropped, and a maximum of 2^k different values can be expressed. The entropy of the matrix also drops considerably, aiding in compression.
- *Quantization*: Similarly, PAS supports quantization using k bits, $k \leq 8$, where 2^k possible values are allowed. The quantization can be done in random manner or uniform manner by analyzing the distribution, and a coding table is used to maintain the integer codes stored in the matrices in PAS. This is most useful for snapshots whose weights are primarily used for fine-tuning or initialization.

The float point schemes present here are not new, and are used in DNN systems in practice [14], [15], [16]. As a lifecycle management tool, PAS lets experienced users select schemes rather than deleting snapshots due to resource constraints. Our evaluation shows storage/accuracy tradeoffs of these schemes.

Bytewise Segmentation for Float Matrices: High entropy of float numbers makes them very hard to compress; compression ratios shown in related work for scientific float point datasets, e.g., simulations, is very low. State of the art compression schemes do not work well for DNN parameters either (Sec. VI). By exploiting DNN low-precision tolerance, we adopt bytewise decomposition from prior work [17], [18] and extend it to our context to store the float matrices. The basic idea is to separate the high-order and low-order mantissa bits, and so a float matrix is stored in multiple chunks; the first chunk consists of 8 high-order bits, and the rest are segmented one byte per chunk. One major advantage is the high-order bits

[2]We do not make a distinction about the bias weight; the typical linear transformation $W'x + b$ is treated as $W \cdot (x, 1) = (W', b)^T \cdot (x, 1)$.



(a) Matrix Storage Graph (b) Optimal Plan without Constraints (c) Optimal Plan with $C_r^{\psi_i}(s_1) \leq 3 \wedge C_r^{\psi_i}(s_2) \leq 6$

Fig. 5. Illustrating Matrix Storage Graph & Plan using a Toy Example

have low entropy, and standard compression schemes (e.g., *zlib*) are effective for them. Apart from its simplicity, the key benefits of segmented approach are two-fold: (a) low-order bytes can be offloaded to remote storage, (b) PAS queries can read high-order bytes only, in exchange for tolerating small errors. Comparison and exploration (`dlv desc`, `dlv diff`) can easily tolerate such errors and, as we show in this paper, `dlv eval` queries can also be made tolerant to these errors.

Delta Encoding across Snapshots: We observed that, due to the non-convexity in training, even re-training the same model with slightly different initializations results in very different parameters. However, the parameters from checkpoint snapshots for the same or similar models tend to be close to each other. Furthermore, across model versions, fine-tuned models generated using fixed initializations from another model often have similar parameters. The observations naturally suggest use of **delta encoding** between checkpointed snapshots in one model version and among latest snapshots across multiple model versions; i.e., instead of storing all matrices in entirety, we can store some in their entirety and others as differences from those. Two possible delta functions (denoted $\ominus$) are *arithmetic subtraction* and *bitwise XOR*[3]. We find the compression footprints when applying the diff $\ominus$ in different directions are similar. We study the delta operators on real models in Sec. V.

C. Optimal Parameter Archival Storage

Given the above background, we next address the question of how to best store a collection of model versions, so that the total storage footprint occupied by the large segmented float matrices is minimized while the retrieval performance is not compromised. This recreation/storage tradeoff sits at the core of any version control system. In recent work [13], the authors study six variants of this problem, and show the NP-hardness of most of those variations. However, their techniques cannot be directly applied in PAS, primarily because their approach is not able to handle the *group retrieval (co-usage)* constraints.

We first introduce the necessary notation, discuss the differences from prior work, and present the new techniques we developed for PAS. In Fig. 4, a *model version* $v \in V$ consists of time-ordered checkpointed *snapshots*, $S_v = s_1, ..., s_n$. Each *snapshot*, s_i consists of a named list of float matrices $M_{v,i} = \{m_k\}$ representing the learned parameters. All matrices in a repository, $\mathcal{M} = \bigcup_{v \in V} \bigcup_{s_i \in S_v} M_{v,i}$, are the parameter artifacts to archive. Each matrix $m \in \mathcal{M}$ is either stored directly, or is recovered through another matrix $m' \in \mathcal{M}$ via a delta operator $\ominus$, i.e. $m = m' \ominus d$, where d is the delta computed using one of the techniques discussed above. In the latter case, the matrix d is stored instead of m. To unify the two cases, we introduce a empty matrix ν_0, and define $\forall \ominus \forall m \in \mathcal{M}, m \ominus \nu_0 = m$.

Definition 1 (Matrix Storage Graph): Given a repository of model versions V, let ν_0 be an empty matrix, and $\mathcal{V} = \mathcal{M} \cup \{\nu_0\}$ be the set of all parameter matrices. We denote by $\mathcal{E} = \{m_i \ominus m_j\} \cup \{m_i \ominus \nu_0\}$ the available deltas between all pairs of matrices. Abusing notation somewhat, we also treat $\mathcal{E}$ as the set of all *edges* in a graph where $\mathcal{V}$ are the vertices. Finally, let $\mathcal{G}_V(\mathcal{V}, \mathcal{E}, c_s, c_r)$ denote the *matrix storage graph* of V, where edge weights $c_s, c_r : \mathcal{E} \mapsto \mathbb{R}^+$ are storage cost and recreation cost of an edge respectively.

Definition 2 (Matrix Storage Plan): Any connected subgraph of $\mathcal{G}_V(\mathcal{V}, \mathcal{E})$ is called a matrix storage plan for V, and denoted by $\mathcal{P}_V(\mathcal{V}_P, \mathcal{E}_P)$, where $\mathcal{V}_P = \mathcal{V}$ and $\mathcal{E}_P \subseteq \mathcal{E}$.

Example 1: In Fig. 5(a), we show a matrix storage graph for a repository with two snapshots, $s_1 = \{m_1, m_2\}$ and $s_2 = \{m_3, m_4, m_5\}$. The weights associated with an edge $e = (\nu_0, m_i)$ reflect the cost of materializing the matrix m_i and retrieving it directly. On the other hand, for an edge between two matrices, e.g., $e = (m_2, m_5)$, the weights denote the storage cost of the corresponding delta and the recreation cost of applying that delta. In Fig. 5(b) and 5(c), two matrix storage plans are shown.

For a matrix storage plan $\mathcal{P}_V(\mathcal{V}_P, \mathcal{E}_P)$, PAS stores all its edges and is able to recreate any matrix m_i following a path starting from ν_0. The *total storage cost* of $\mathcal{P}_V$, denoted as $C_s(\mathcal{P}_V)$, is simply the sum of edge storage costs, i.e. $C_s(\mathcal{P}_V) = \sum_{e \in \mathcal{E}_P} c_s(e)$. Computation of the *average snapshot recreation cost* is more involved and depends on the retreival scheme used:

- *Independent* scheme recreates each matrix m_i one by one by following the shortest path (Υ_{ν_0, m_i}) to m_i from ν_0. In that case, the recreation cost is simply computed by summing the recreation costs for all the edges along the shortest path.
- *Parallel* scheme accesses all matrices of a snapshot in parallel (using multiple threads); the longest shortest path from ν_0 defines the recreation cost for the snapshot.
- *Reusable* scheme considers caching deltas on the way, i.e., if paths from ν_0 to two different matrices overlap, then the shared computation is only done once. In that case, we need to construct the lowest-cost *Steiner tree* ($\mathcal{T}_{\mathcal{P}_V, s_i}$) involving ν_0 and the matrices in the snapshot. However, because multiple large matrices need to be kept in memory simultaneously, the memory consumption of this scheme can be large.

PAS can be configured to use any of these options during the actual query execution. However, solving the storage optimization problem with *Reusable* scheme is nearly impossible; since the Steiner tree problem is NP-Hard, just computing the cost of a solution becomes intractable making it hard to even compare two different storage solutions. Hence, during the storage optimization process, PAS can only support *Independent* or *Parallel* schemes.

Retrieval Scheme	*Recreation* $C_r^{\psi}(\mathcal{P}_V, s_i)$	*Solution of Prob.1*
Independent (ψ_i)	$\sum_{m_j \in s_i} \sum_{e_k \in \Upsilon_{\nu_0, m_j}} c_r(e_k)$	Spanning tree
Parallel (ψ_p)	$\max_{m_j \in s_i} \{\sum_{e_k \in \Upsilon_{\nu_0, m_j}} c_r(e_k)\}$	Spanning tree
Reusable (ψ_r)	$\sum_{e_k \in \mathcal{T}_{\mathcal{P}_V, s_i}} c_r(e_k)$	Subgraph

TABLE III. Recreation Cost of a Snapshot s_i $C_r(\mathcal{P}_V, s_i)$ in a plan $\mathcal{P}_V$

[3] Delta functions for matrices with different dimensions are discussed in the long version of the paper; techniques in Sec IV work with minor modifications.

In the example above, the edges are shown as being undirected indicating that the deltas are symmetric. In general, we allow for directed deltas to handle asymmetric delta functions, and also for multiple directed edges between the same two matrices. The latter can be used to capture different options for storing the delta; e.g., we may have one edge corresponding to a remote storage option, where the storage cost is lower and the recreation cost is higher; whereas another edge (between the same two matrices) may correspond to a local SSD storage option, where the storage cost is the highest and the recreation cost is the lowest. Our algorithms can thus automatically choose the appropriate storage option for different deltas.

Similarly, PAS is able to make decisions at the level of byte segments of float matrices, by treating them as separate matrices that need to be retrieved together in some cases, and not in other cases. This, combined with the ability to incorporate different storage options, is a powerful generalization that allows PAS to make decisions at a very fine granularity.

Given this notation, we can now state the problem formally. Since there are multiple optimization metrics, we assume that constraints on the retrieval costs are provided and ask to minimize the storage.

Problem 1 (Optimal Parameter Archival Storage): Given a matrix storage graph $\mathcal{G}_V(\mathcal{V}, \mathcal{E}, c_s, c_r)$, let θ_i be the *snapshot recreation cost budget* for each $s_i \in S$. Under a retrieval scheme ψ, find a matrix storage plan $\mathcal{P}_V{}^*$ that minimizes the *total storage cost*, while satisfying recreation constraints, i.e.:
minimize$_{\mathcal{P}_V} C_s(\mathcal{P}_V)$; s.t. $\forall s_i \in S, C_r{}^{\psi}(\mathcal{P}_V, s_i) \leq \theta_i$

Example 2: In Fig. 5(b), without any recreation constraints, we show the best storage plan, which is the minimum spanning tree based on c_s of the matrix storage graph, $C_s(\mathcal{P}_V) = 19$. Under independent scheme ψ_i, $C_r{}^{\psi_i}(\mathcal{P}_V, s_1) = 3$ and $C_r{}^{\psi_i}(\mathcal{P}_V, s_2) = 7.5$. In Fig. 5(c), after adding two constraints $\theta_1 = 3$ and $\theta_2 = 6$, we shows an optimal storage plan $\mathcal{P}_V{}^*$ satisfying all constraints. The storage cost increases, $C_s(\mathcal{P}_V{}^*) = 24$, while $C_r{}^{\psi_i}(\mathcal{P}_V{}^*, s_1) = 3$ and $C_r{}^{\psi_i}(\mathcal{P}_V{}^*, s_2) = 6$.

Although this problem variation might look similar to the ones considered in recent work [13], none of the variations studied there can handle the co-usage constraints (i.e., the constraints on simultaneously retrieving a group of versioned data artifacts). One way to enforce such constraints is to treat the entire snapshot as a single data artifact that is stored together; however, that may force us to use an overall suboptimal solution because we would not be able to choose the most appropriate delta at the level of individual matrices. Another option would be to sub-divide the retrieval budget for a snapshot into constraints on individual matrices in the snapshot. As our experiments show, that can lead to significantly higher storage utilization. Thus the formulation above is a strict generalization of the formulations considered in that prior work. The proofs of the following two theorems can be found in the extended version of the paper.

Theorem 1: Optimal Parameter Archival Storage Problem is NP-hard for all retrieval schemes in Table III.

Lemma 2: The optimal solution for Problem 1 is a spanning tree when retrieval scheme ψ is *independent* or *parallel*.

The above lemma is not true for the *reusable* scheme (ψ_r); snapshot Steiner trees satisfying different recreation constraints may share intermediate nodes resulting in a subgraph solution.

Constrained Spanning Tree Problem: In Problem 1, storage cost minimization while ignoring the recreation constraints leads to a minimum spanning tree (MST) of the matrix storage graph; whereas the snapshot recreation constraints are best satisfied by using shortest path trees (SPT). These problems are often referred to as constrained spanning tree problems [19] or shallow-light tree constructions [20], which have been studied in areas other than dataset versioning, such as VLSI designs. Khuller et al. [21] propose an algorithm called LAST to construct such a "balanced" spanning tree in an undirected graph G. LAST starts with a minimum spanning tree of the provided graph, traverses it in a DFS manner, and adjusts the tree by changing parents to ensure the path length in constructed solution is within $(1+\epsilon)$ times of shortest path in G, i.e. $C_r(T, v_i) \leq (1+\epsilon) C_r(\Upsilon_{v_0,v_i}, v_i)$, while total storage cost is within $(1+\frac{2}{\epsilon})$ times of MST. In our problem, the co-usage constraints of matrices in each snapshot form hyperedges over the matrix storage graph making the problem more difficult.

In the rest of the discussion, we adapt meta-heuristics for constrained MST problems to develop two algorithms: the first one (PAS-MT) is based on an iterative refinement scheme, where we start from an MST and then adjust it to satisfy constraints; the second one is a priority-based tree construction algorithm (PAS-PT), which adds nodes one by one and encodes heuristic in the priority function. Both algorithms aim to solve the parallel and independent recreation schemes, and thus can also find feasible solution for reusable scheme. Due to large memory footprints of intermediate matrices, we leave improving reusable scheme solutions for future work.

PAS-**MT:** The algorithm starts with T as the MST of $\mathcal{G}_V(\mathcal{V}, \mathcal{E})$, and iteratively adjusts T to satisfy the broken snapshot recreation constraints, $U = \{s_i | C_r(T, s_i) > \theta_i\}$, by swapping one edge at a time. We denote p_i as the parent of v_i, $(p_i, v_i) \in T$ and $p_0 = \phi$, and successors of v_i in T as $\mathcal{D}_i$. A *swap operation* on (p_i, v_i) to $(v_s, v_i) \in \mathcal{E} - T \wedge v_s \notin \mathcal{D}_i$ changes parent of v_i to v_s.

Lemma 3: A swap operation on v_i changes storage cost of $C_s(T)$ by $c_s(p_i, v_i) - c_s(v_s, v_i)$, and changes recreation costs of v_i and its successors $\mathcal{D}_i$ by: $C_r(T, v_i) - C_r(T, v_s) - c_r(v_s, v_i)$.

The proof can be derived from definition of C_s and C_r by inspection. When selecting edges in $\mathcal{E} - T$, we choose the one which has the largest marginal gain for unsatisfied constraints:

$$\psi_i : \max_{(v_s,v_i)\in\mathcal{E}-T\wedge v_s\notin\mathcal{D}_i} \{\frac{\sum_{s_k\in U}\sum_{v_j\in s_k\cap\mathcal{D}_i}(C_r(T,v_i) - C_r(T,v_s) - c_r(v_s,v_i))}{c_s(v_s,v_i) - c_s(p_i,v_i)}\} \quad (1)$$

$$\psi_p : \max_{(v_s,v_i)\in\mathcal{E}-T\wedge v_s\notin\mathcal{D}_i} \{\frac{\sum_{s_k\in U}(C_r(T,v_i) - C_r(T,v_s) - c_r(v_s,v_i))}{c_s(v_s,v_i) - c_s(p_i,v_i)}\} \quad (2)$$

The actual formula used is somewhat more complex, and handles non-positive denominators. Eq. 1 sums the gain of recreation cost changes among all matrices in the same snapshot s_i (for the independent scheme), while Eq. 2 uses the max change instead (for the *parallel* scheme).

The algorithm iteratively swaps edges and stops if all recreation constraints are satisfied or no edge returns a positive gain. A single step examines $|\mathcal{E} - T|$ edges and $|U|$ unsatisfied constraints, and there are at most $|\mathcal{E}|$ steps. Thus the complexity

is bounded by $O(|\mathcal{E}|^2|S|)$. More sophisticated scheme uses Fibonacci heap to incrementally update the edge gains.

PAS-**PT:** This algorithm constructs a solution by "growing" a tree starting with an empty tree. The algorithm examines the edges in $\mathcal{G}_V(\mathcal{V}, \mathcal{E})$ in the increasing order by the storage cost c_s; a priority queue is used to maintain all the candidate edges and is populated with all the edges from v_0 in the beginning. At any point, the edges in Q are the ones that connect a vertex in T, to a vertex outside T. Using an edge $e_{ij} = (v_i, v_j)$ (s.t., $v_i \in V_T \wedge v_j \in \mathcal{V} - V_T$) popped from Q, the algorithm tries to add v_j to T with minimum storage increment $c_s(e_{ij})$. Before adding v_j, it examines whether the constraints of affected groups s_a (s.t., $v_j \in s_a$) are satisfied using actual and estimated recreation costs for vertices $\{v_k \in s_a\}$ in V_T and $\mathcal{V} - V_T$ respectively; if $v_k \in V_T$, actual recreation cost $C_r(T, v_k)$ is used, otherwise the lower bound of it, i.e. $c_r(v_0, v_k)$ is used as an estimation.

Once an edge e_{ij} is added to T, the inner edges $\mathcal{I}_T^j = \{(v_k, v_j) | v_k \in V_T\}$ of newly added v_j are dequeued from Q, while the outer edges $\mathcal{O}_T^j = \{(v_j, v_k) \mid v_k \in \mathcal{V} - V_T\}$ are enqueued. If the storage cost of existing vertices in T can be improved (i.e. $C_s(T, v_k) > c_s(v_k, v_j)$), and recreation cost is not more (i.e. $C_r(T, v_k) \geq C_r(T, v_j) + c_r(v_k, v_j)$), then the parent p_k of v_k in T is replaced to v_j via the swap operation, decreasing the storage but not increasing affected group recreation cost.

The algorithm stops if Q is empty and T is a spanning tree. In the case when Q is empty but $V_T \subset \mathcal{V}$, an *adjustment operation* on T to increase storage cost and satisfy the group recreation constraints is performed. For each $v_u \in \mathcal{V} - V_T$, we append it to v_0, then in each unsatisfied group s_i that v_u belongs to, optimally, we want to choose a set of $\{v_g\} \subseteq s_i \cap T$ to change their parents in T, such that the decrement of storage cost is minimized while recreation cost is satisfied. The optimal adjustment itself can be viewed as a knapsack problem with extra non-cyclic constraint of T, which is NP-hard. Instead, we use the same heuristic in Eq. 1 to adjust $v_g \in s_i \cap T$ one by one by swapping its parent p_g to v_s until the group constraints cannot improved. Similarly, the parallel scheme ψ_p uses Eq. 2 for the adjustment operation. The complexity of this algorithm is $O(|\mathcal{E}|^2|S|)$.

D. Model Evaluation Scheme in PAS

Model evaluation, i.e., applying a DNN forward on a data point to get the prediction result, is a common task to explore, debug and understand models. Given a PAS storage plan, a `dlv eval` query requires uncompressing and applying deltas along the path to the model. We develop a novel model evaluation scheme utilizing the segmented design, that progressively accesses the low-order segments only when necessary, and guarantees no errors for arbitrary data points.

The basic intuition is that: when retrieving segmented parameters, we know the minimum and maximum values of the parameters (since higher order bytes are retrieved first). If the prediction result is the same for the entire range of those values, then we do not need to access the lower order bytes. However, considering the high dimensions of parameters, non-linearity of the DNN model, unknown full precision value when issuing the query, it is not clear if this is feasible.

We define the problem formally, and illustrate the determinism condition that we use to develop our algorithm. Our technique is inspired from theoretical stability analysis in numerical analysis. We make the formulation general to be applicable to other prediction functions. The basic assumption is that the prediction function returns a vector showing relative strengths of the classification labels, then the dimension index with the maximum value is used as the predicted label.

Problem 2 (Parameter Perturbation Error Determination): Given a prediction function $\mathcal{F}(d, W) : \mathbb{R}^m \times \mathbb{R}^n \mapsto \mathbb{R}^c$, where d is the data and W are the learned weights, the prediction result c_d is the dimension index with the highest value in the output $o \in \mathbb{R}^c$. When W value is uncertain, i.e., each $w_i \in W$ in known to be in the range $[w_{i,\min}, w_{i,\max}]$, determine whether c_d can be ascertained without error.

When W is uncertain, the output o is uncertain as well. However, if we can bound the individual entries in o, Lemma 4 is an applicable necessary condition for determining error:

Lemma 4: Let $o_i \in o$ vary in range $[o_{i,\min}, o_{i,\max}]$. If $\exists k$ such that $\forall i,\ o_{k,\min} > o_{i,\max}$, then prediction result c_d is k.

Next we illustrate a query procedure that, given data d, evaluates a DNN with weight perturbations and determines the output perturbation on the fly. Recall that DNN is a nested function (Sec. II), we derive the output perturbations when evaluating a model while preserving perturbations step by step:

$$x_{0,k}^{\min} = \sum_j \min\{W_{0,k,j} d_j\} + \min\{b_{0,k}\}; \quad x_{0,k}^{\max} = \sum_j \max\{W_{0,k,j} d_j\} + \max\{b_{0,k}\}$$

Next, activation function σ_0 is applied. Most of the common activation functions are monotonic functions: $\mathbb{R} \mapsto \mathbb{R}$, (e.g. sigmoid, ReLu), while pool layer functions are min, max, avg functions over several dimensions. It is easy to derive the perturbation of output of the activation function, $[f_{0,k,\min}, f_{0,k,\max}]$. During the evaluation query, instead of 1-D actual output, we carry 2-D perturbations, as the actual parameter value is not available. Nonlinearity decreases or increases the perturbation range. Now the output perturbation at f_i can be calculated similarly, except now both W and f_{i-1} are uncertain:

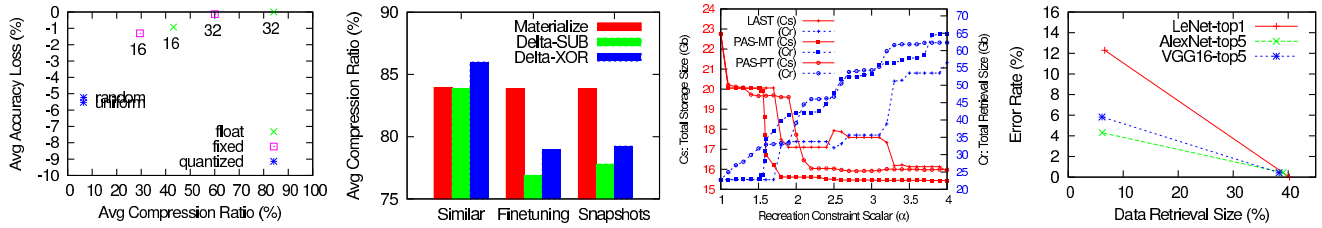
$$x_{i,k}^{\min} = \sum_j \min\{W_{i,k,j} f_{i-1,j}\} + \min\{b_{i,k}\}; \quad x_{i,k}^{\max} = \sum_j \max\{W_{i,k,j} f_{i-1,j}\} + \max\{b_{i,k}\}$$

Applying these steps iteratively until last layer, we can then apply Lemma 4, the condition of error determinism, to check if the result is correct. If not, then lower order segments of the float matrices are retrieved, and the evaluation is re-performed.

This progressive evaluation query techniques dramatically improve the utility of PAS, as we further illustrate in our experimental evaluation. Note that, other types of queries, e.g., matrix plots, activation plots, visualizations, etc., can often be executed without retrieving the lower-order bytes either.

V. Evaluation Study

ModelHub is designed to work with a variety of deep learning backends; our current prototype interfaces with `caffe` [3] through a wrapper that can extract caffe training logs, and read and write parameters for training. We have also built a custom layer in `caffe` to support progressive queries. The `dlv` command-line suite is implemented as a Ruby gem, utilizing `git` as internal VCS and `sqlite3` and PAS as backends to manage the set of heterogeneous artifacts in the local client. PAS is built in C++ with gcc 5.4.0. All experiments are conducted on a Ubuntu Linux 16.04 machine with an 8-core 3.0GHz AMD FX-380 processor, 16GB memory, and



(a) Compression-Accuracy Tradeoff for Float Representation Schemes

(b) Compression Performance for Different Delta Schemes & Models

(c) Comparing PAS Archival Storage Algorithms for SD

(d) Progressive Evaluation Query Processing Using High-Order Bytes

Fig. 6. Evaluation Results for PAS

NVIDIA GTX 970 GPU. We use zlib for compression; unless specifically mentioned, the compression level is set to 6. When wrapping and modifying caffe, the code base version is rc3.

In this section, we present a comprehensive evaluation with real-world and synthetic datasets aimed at examining our design decisions, differences of configurations in PAS, and performance of archiving and progressive query evaluation techniques proposed in earlier sections.

A. Dataset Description

Real World Dataset: To study the performance of PAS design decisions, we use a collection of shared caffe models published in caffe repository or Model Zoo. In brief, LeNet-5 [7] is a convolutional DNN with 431k parameters. The reference model has 0.88% error rate on MNIST. AlexNet [8] is a medium-sized model with 61 million parameters, while VGG-16 [9] has 1.9 billion parameters. Both AlexNet and VGG-16 are tested on ILSVRC-2012 dataset. The downloaded models have 43.1%, and 31.6% top-1 error rate respectively. Besides, to study the delta performance on model repositories under different workloads (i.e., retraining, fine-tuning): we use VGG-16/19 and CNN-S/M/F [22], a set of similar models developed by VGG authors to study model variations. They are similar to VGG-16, and retrained from scratch; for fine-tuning, we use VGG-Salient [23] a fine-tuning VGG model which only changes last full layer.

Synthetic Datasets: Lacking sufficiently fine-grained real-world repositories of models, to evaluate performance of parameter archiving algorithms, we developed an automatic modeler to enumerate models and hyperparameters to produce a `dlv` repository. We generated a synthetic dataset (SD): simulating a modeler who is enumerating models to solve a face recognition task, and fine-tuning a trained VGG. SD results in similar DNNs and relatively similar parameters across the models. Based on real trained dataset SD, we vary the delta ratios, group sizes, and number of models to derive a collection of repositories (RD). The datasets are shared online[4].

To elaborate, the automation is driven by a state machine that applies modeling practices from the real world. The modeler updates the VGG network architecture slightly and changes VGG object recognition goal to a face prediction task (prediction labels changed from 1000 to 100, so the last layer is changed); various fine-tuning hyperparameter alternations are applied by mimicking practice [24]. SD in total has 54 model versions, each of which has 10 snapshots. A snapshot has 16 parametric layers and a total of 1.96×10^{10} floats.

[4]Dataset Details: http://www.cs.umd.edu/~hui/code/modelhub

B. Evaluation Results

Float Representation & Accuracy: We show the effect of different float encoding schemes on compression and accuracy in Fig. 6(a); this is a tradeoff that the user often needs to consider when configuring ModelHub to save a model. In Fig. 6(a), for each scheme, we plot the average compression ratio versus the average accuracy drop when applying PAS float schemes on the three real-world models. Here, *random* and *uniform* denote two standard quantization schemes. As we can see, we can get very high compression ratios (a factor of 20 or so) without a significant loss in accuracy, which may be acceptable in many scenarios.

Delta Encoding & Compression Ratio Gain: Next we study the usefulness of delta encoding in real-world models in the following scenarios: **a)** *Similar*: latest snapshots across similar models (CNN-S/M/F, VGG-16); **b)** *Fine-tuning*: fine-tuning models (VGG-16, VGG-Salient); and **c)** *Snapshots*: snapshots for the same VGG models in SD between iterations. In Fig. 6(b), for different delta schemes, namely, storing original matrices (*Materialize*), arithmetic subtraction (*Delta-SUB*), and bitwise XOR diff (*Delta-XOR*), the comparison is shown (i.e., we show the results of compressing the resulting matrices using **zlib**). The figure shows the numbers under lossless compression scheme (float 32), which has the largest storage footprint.

As we can see, delta scheme is not always good, due to the non-convexity and high entropy of parameters. For models under similar architectures, storing materialized original parameters is often better than applying delta encoding. With fine-tuning and nearby snapshots, the delta is always better, and arithmetic subtraction is consistently better than bitwise XOR. We saw similar results for many other models. These findings are useful for PAS implementation decisions, where we only perform delta between nearby snapshots in a single model, or for the fine-tuning setting among different models.

Table IV shows the delta encoding results when using two lossy schemes, fixed point conversion and *normalization*, for fine-tuned VGG datasets, but without reducing the number of bits used (i.e., we still use 32 bits to store the numbers). Normalization refers to adding a sufficiently large number to all the floats so that the radixes and the signs are aligned, whereas fixed point conversion uses a single exponent for all the numbers in a matrix. As we can see, for both of these, delta encoding can result in sigificant gains. Introducing additional lossiness, e.g., through using fewer bits, further improves the performance, but at the expense of significantly higher accuracy loss.

Schemes	Configuration	Materialize	Delta-SUB
Float Number Representation	Lossless	92.83%	86.39%
	Lossless, bytewise	83.85%	76.89%
	Fix point	72.43%	57.15%
	Fix point, bytewise	58.68%	49.34%
After Normalization	Lossless	68.06%	47.69%
	Lossless, bytewise	56.15%	**36.60%**
	Fix point	69.11%	48.94%
	Fix point, bytewise	55.36%	36.88%

TABLE IV. Delta Performance For Lossless & Lossy Schemes, 32-bits

Optimal Parameter Archival Storage: Fig. 6(c) shows the results of comparing PAS-PT, PAS-MT and the baseline LAST [21] for the optimal parameter archival problem. Using dataset SDand RD, we derive nearby snapshot deltas as well as model-wise deltas among the latest snapshots. To compare with LAST clearly, we vary the recreation threshold using a scalar α to mimic a full precision archiving problem instance with different constraints, i.e., $C_r(T, s_i) \leq \alpha \cdot C_r(\text{SPT}, s_i)$. The SPT for SD is 22.77Gb and the MST is 15.44Gb. In Fig. 6(c), the left y-axis denotes the storage cost (C_s) while the right y-axis is the recreation cost (C_r). As we can see, in most cases, PAS-MT and PT find much better storage solutions that are very close to the MST (the best possible) by exploiting the recreation thresholds. In contrast, LAST, which cannot handle group constraints, returns worse storage plans and cannot utilize the recreation constraints fully. Between MT and PT, since MT starts from the MST and adjusts it, when the constraints are tight (i.e., $\alpha < 1.5$), MT cannot alter it to very different trees and the recreation constraints are underutilized; however, PT can exploit the constraints when selecting edges to grow the tree. On the other hand, when the threshold is loose ($\alpha \in [1.5, 2]$), MT's edge swapping strategy is able to refine MST extensively, while PT prunes edges early and cannot find solutions close to MST. When the constraints continue to loosen, both PAS algorithms find good plans, while LAST can only do so at very late stages ($\alpha > 3$). In practice, the best option might be to execute both algorithms and pick the best solution for a given setting.

Retrieval Performance: Next we show the retrieval performance for PAS storage plans using the SD dataset. The main query type of interest is snapshot retrieval, which would retrieve all segments of a snapshot or, for a partial retrieval query, the high-order segments. In Table V, the average recreation time of a snapshot for a moderate PAS storage plan ($\alpha = 1.6$) is compared with the two extreme cases, full materialization (SPT), and best compression without recreation constraints (MST). As we can see, PAS is not only able to find good storage solutions which satisfy recreation constraints, but also supports flexible query access schemes. Under partial access of high order bytes, the query times for segmented snapshots are better than uncompressing the fully materialized model.

Progressive Query Evaluation: We study the efficiency of the progressive evaluation technique using perturbation error determination scheme on real-world models (LeNet, AlexNet, VGG16) and their corresponding datasets. The original parameters are 4-byte floats, which are archived in segments in PAS. We modify `caffe` implementation of involved layers and pass two additional blobs (min/max errors) between layers. The perturbation error determination algorithm uses high order segments, and answers `eval` query on the test dataset. The algorithm determines whether top-k (1 or 5) result needs lower order bytes (i.e., matched index value range overlaps with $k+1$ index value range). The result is summarized in Fig. 6(d). The y-axis shows the error rate. The x-axis shows the percentage of data that needs to be retrieved (i.e., 2 bytes or 1 byte per float). As one can see, the prediction errors requiring full precision lower-order bytes are very small. The less high-order bytes used, higher the chance of potential errors. The consistent result of progressive query evaluation on real models supports our design decision of segmented float storage.

Storage Plan	Query	Independent (s)	Parallel (s)
Materialization	Full	3.49	2.16
Min Storage	Full	8.47	4.85
PAS ($\alpha = 1.6$)	Full	8.1	4.59
	2 bytes	3.19	0.38
	1 byte	1.60	0.18

TABLE V. Recreation Performance Comparison of Storage Plans

VI. Related Work

Machine Learning Systems: There have been several high-profile deep learning systems in recent years, but those typically focus on the training aspects (e.g., on distributed training, how to utilize GPUs or allow symbolic formulas, etc.) [3], [4], [25], [26], [27]. The data 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, including pushing predictive models into databases [28], [29], accelerating tasks using database optimizing methods [2], [30], and managing modeling lifecycles and serving predictive models in advanced ways [31], [32]. ModelHub is motivated by similar principles; aside from a focus on DNNs, it also supports *versioning* as a first-class construct [11] which differentiates it from that work.

DNN Manipulation Frameworks: There are several popular model manipulation libraries and methods for major training systems, including *Keras* [33] for Theano and Tensorflow, and *nngraph* for Torch. Those usually target limited aspects of the overall modeling lifecycle, and usually support a procedural approach to create and manipulate models. This makes it hard to reuse a collection of existing models with different properties, to maintain lineages among models, or to cleanly interleave the different steps of querying, mutation, and evaluation. The difference between frameworks like Keras and `DQL` is somewhat akin to the difference between MapReduce and SQL; we believe that using a more structured and abstract language like `DQL` makes it easier to interact with and explore models. On the flip side, `DQL` relies on a data model that is built on pre-defined layers instead of tensor operations, which limits its ability to make fine-grained network adjustments, such as changing layer behaviors. We note that `dlv` does not preclude users from continuing to use Keras or similar tools to construct models, and artifacts created by them are treated the same as those created through `DQL`.

DNN Compression: There has been increasing interest on compressing DNN models, motivated in part by the need

to deploy them on devices with simple instruction sets, low memory, and/or energy constraints [15], [34], [35]. However, the goal of those works is to simplify the model in a lossy manner with as little loss of accuracy as possible, which makes that work orthogonal to the archival problem we face in ModelHub; in fact, simplified models are likely to compress much better, magnifying the gains of our approach as our experimental results show. Further, these methods often require heavy retraining or expensive computations (*k*-means, SVD, etc.) to derive simpler models, which makes them too heavy-weight in an interactive setting for which `DLV` is designed. TensorFlow also uses compression during training to reduce communication overhead, but does not consider the problem of compressing models jointly for minimizing storage footprint.

DNNs with Low Precision Floats: Low precision floats have been exploited in accelerating training and testing [14], [35], [36]; those works present techniques and empirical results when training and testing DNNs with limited precision. ModelHub differs in its focus on parameter archiving, and answering lifecycle management queries.

Stability Analysis Results Stability analysis of DNNs, with primary focus on statistical measures of stability, is a well-studied concept [37], [38], [39]. ModelHub uses basic perturbation analysis techniques and focuses on novel progressive query answering in a segmented float storage.

VII. Conclusion and Future Work

In this paper, we described some of the key data management challenges in learning, managing, and adjusting deep learning models, and presented our ModelHub system that attempts to address those challenges in a systematic fashion. The goals of ModelHub are multi-fold: (a) to make it easy for a user to explore the space of potential models by tweaking the network architecture and/or the hyperparameter values, (b) to minimize the burden in keeping track of the metadata including the accuracy scores and the fine-grained results, and (c) to compactly store a large number of models and constituent snapshots without compromising on query or retrieval performance. We presented several high-level abstractions, including a command-line version management tool and a domain-specific language, for addressing the first two goals. Anecdotal experience with our early users suggests that both of those are effective at simplifying the model exploration tasks. We also developed a read-optimized parameter archival storage for storing the learned weight parameters, and designed novel algorithms for storage optimization and for progressive query evaluation. Extensive experiments on real-world and synthetic models verify the design decisions we made and demonstrate the advantages of proposed techniques.

Acknowledgements: This work was supported in part by NSF under grants 1513972 and 1513443, and in part by the Office of Naval Research under grant N000141612713 entitled 'Visual Common Sense Reasoning for Multi-agent Activity Prediction and Recognition.'

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, 2015.
[2] C. Zhang, A. Kumar, and C. Ré, "Materialization optimizations for feature selection workloads," in *SIGMOD*, 2014.
[3] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *ACM MM*, 2014.
[4] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *OSDI*, 2016.
[5] R. Jozefowicz, W. Zaremba, and I. Sutskever, "An empirical exploration of recurrent network architectures," in *ICML*, 2015.
[6] H. Miao, A. Chavan, and A. Deshpande, "ProvDB: A system for lifecycle management of collaborative analysis workflows," *arXiv preprint arXiv:1610.04963*, 2016.
[7] Y. LeCun *et al.*, "Handwritten digit recognition with a back-propagation network," in *NIPS*, 1990.
[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.
[9] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.
[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.
[11] A. Bhardwaj *et al.*, "DataHub: Collaborative data science and dataset version management at scale," in *CIDR*, 2015.
[12] H. Miao *et al.*, "ModelHub: Towards unified data and lifecycle management for deep learning," *arXiv preprint arXiv:1611.06224*, 2016.
[13] S. Bhattacherjee *et al.*, "Principles of dataset versioning: Exploring the recreation/storage tradeoff," *PVLDB*, 2015.
[14] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on CPUs," in *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.
[15] S. Han *et al.*, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *ICLR*, 2016.
[16] M. Courbariaux *et al.*, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.
[17] E. R. Schendel *et al.*, "Isobar preconditioner for effective and high-throughput lossless data compression," in *ICDE*, 2012.
[18] S. Bhattacherjee, A. Deshpande, and A. Sussman, "Pstore: an efficient storage framework for managing scientific data," in *SSDBM*, 2014.
[19] N. Deo and N. Kumar, "Computation of constrained spanning trees: A unified approach," in *Network Optimization*, 1997.
[20] A. B. Kahng and G. Robins, *On optimal interconnections for VLSI*. Springer Science & Business Media, 1994, vol. 301.
[21] S. Khuller, B. Raghavachari, and N. Young, "Balancing minimum spanning trees and shortest-path trees," *Algorithmica*, 1995.
[22] K. Chatfield *et al.*, "Return of the devil in the details: Delving deep into convolutional nets," in *BMVC*, 2014.
[23] J. Zhang, S. Ma, M. Sameki, S. Sclaroff, M. Betke, Z. Lin, X. Shen, B. Price, and R. Mech, "Salient object subitizing," in *CVPR*, 2015.
[24] R. Girshick, "Fast R-CNN," in *ICCV*, 2015.
[25] J. Dean *et al.*, "Large scale distributed deep networks," in *NIPS*, 2012.
[26] T. Chilimbi *et al.*, "Project ADAM: Building an efficient and scalable deep learning training system," in *OSDI*, 2014.
[27] W. Wang *et al.*, "Singa: Putting deep learning in the hands of multimedia users," in *ACM MM*, 2015.
[28] M. Akdere *et al.*, "The case for predictive database systems: Opportunities and challenges." in *CIDR*, 2011.
[29] X. Feng, A. Kumar, B. Recht, and C. Ré, "Towards a unified architecture for in-rdbms analytics," in *SIGMOD*, 2012.
[30] A. Kumar, J. Naughton, and J. M. Patel, "Learning generalized linear models over normalized data," in *SIGMOD*, 2015.
[31] M. Vartak *et al.*, "Supporting fast iteration in model building," in *LearningSys*, 2015.
[32] D. Crankshaw, X. Wang, J. E. Gonzalez, and M. J. Franklin, "Scalable training and serving of personalized models," in *LearningSys*, 2015.
[33] F. Chollet, "Keras," https://github.com/fchollet/keras, 2015.
[34] E. L. Denton *et al.*, "Exploiting linear structure within convolutional networks for efficient evaluation," in *NIPS*, 2014.
[35] W. Sung, S. Shin, and K. Hwang, "Resiliency of deep neural networks under quantization," *arXiv preprint arXiv:1511.06488*, 2015.
[36] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *ICML*, 2015.
[37] M. Stevenson *et al.*, "Sensitivity of feedforward neural networks to weight errors," *IEEE Trans. Neural Networks*, vol. 1, 1990.
[38] X. Zeng *et al.*, "Sensitivity analysis of multilayer perceptron to input and weight perturbations," *IEEE Trans. Neural Networks*, vol. 12, 2001.
[39] J. Yang, X. Zeng, and S. Zhong, "Computation of multilayer perceptron sensitivity to input perturbation," *Neurocomputing*, 2013.