MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis

Manasi Vartak
MIT CSAIL
mvartak@csail.mit.edu

Joana M. F. da Trindade
MIT CSAIL
jmf@csail.mit.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

Matei Zaharia
Stanford University
matei@cs.stanford.edu

ABSTRACT

Model diagnosis is the process of analyzing machine learning (ML) model performance to identify where the model works well and where it doesn't. It is a key part of the modeling process and helps ML developers iteratively improve model accuracy. Often, model diagnosis is performed by analyzing different datasets or *intermediates* associated with the model such as the input data and hidden representations learned by the model (e.g., [4, 24, 39]). The bottleneck in fast model diagnosis is the creation and storage of model intermediates. Storing these intermediates requires tens to hundreds of GB of storage whereas re-running the model for each diagnostic query slows down model diagnosis. To address this bottleneck, we propose a system called MISTIQUE that can work with traditional ML pipelines as well as deep neural networks to efficiently capture, store, and query model intermediates for diagnosis. For each diagnostic query, MISTIQUE intelligently chooses whether to re-run the model or read a previously stored intermediate. For intermediates that are stored in MISTIQUE, we propose a range of optimizations to reduce storage footprint including quantization, summarization, and data de-duplication. We evaluate our techniques on a range of real-world ML models in scikit-learn and Tensorflow. We demonstrate that our optimizations reduce storage by up to 110X for traditional ML pipelines and up to 6X for deep neural networks. Furthermore, by using MISTIQUE, we can speed up diagnostic queries on traditional ML pipelines by up to 390X and 210X on deep neural networks.

ACM Reference Format:
Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196934

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196934

1 INTRODUCTION

Machine learning (ML) is the paradigm of "programming using data" [36]. It is the process of taking a model specification (e.g., type of model, model structure), providing training data, and using a training procedure to *learn* model parameters that fit the training data. Since the model is learned from data, it is only natural that diagnosing problems with the model or interpreting the model involves analyses of data artifacts produced during modeling. *Diagnosing* an ML model involves answering questions such as "why does the home price prediction model under-perform on old Victorian homes?" or "why does the image classification model classify a frog as a ship?" Relatedly, model interpretability (e.g., [4, 15, 29, 40]) involves answering questions about how a model works (e.g., "what is the concept learned by a particular neuron?") and why it makes certain predictions. For simplicity, in this paper, we will call both kinds of analyses described above as "model diagnosis."

Diagnostic techniques such as the ones above can be answered by analyzing different data artifacts related to the model including input data, prediction values, and data representations produced by the model (e.g., high-dimensional representations of homes or images learned by the model). We collectively refer to these datasets as *model intermediates* (formal definition in Sec. 2).

Given the importance of model intermediates for diagnosis, in this paper, we explore the question of how to efficiently store and query model intermediates to support efficient model diagnosis. We propose MISTIQUE (**M**odel **I**ntermediate **ST**ore and **QU**ery **E**ngine), a system designed to capture, store, and query model intermediates to support diagnostic queries.

1.1 Motivating Examples

We begin by highlighting three diagnostic techniques that have been proposed in the literature and describe the role that model intermediates play in each of them. A more extensive list of techniques is presented in Sec. 2.2.

Visualizations: A popular means to understand the working of any model is via *visualization*. For example, the ActiVis tool from Facebook [24] (screenshot in Fig.12 in Appendix) provides developers of neural networks an interactive visualization of neuron activations. This information can help ML developers identify activation patterns, compare activations between classes, and find potential sources of error. Similar tools have also been built for traditional modeling pipelines. For example, VizML [13] (Fig.13, Appendix) provides an interface where ML developers can prioritize errors, examine feature distribution, and debug model results.

Intermediates. In order to visualize arbitrary model intermediates, the relevant intermediates must first be generated and stored (re-running the model each time is too expensive for an interactive setting). For ActiVis, this means that data representations at each model layer must be stored. As expected, the total cost to store all intermediates is tremendous. E.g., storing intermediates for ten variants of the popular VGG16 network [46] on a dataset with 50K examples requires 350GB of compressed storage. As a result, ActiVis requires users to restrict the layers and number of examples for which intermediates (and aggregates of intermediates) will be logged.

SVCCA: Raghu et. al. recently proposed SVCCA [39], a technique to compare representations learned by different layers in one or more neural networks. In brief, SVCCA takes as input the activations of neurons in two layers l_1 and l_2, performs SVD on the two sets of activations, projects activations into the subspace identified by SVD, and computes canonical correlation to find directions in these subspaces that are most aligned (see Alg. 1 in Appendix). Intermediates. To perform class sensitivity analyses across the whole network as described [39], activations for *all* examples at *all* layers must be available. Furthermore, if one wants to study training procedure dynamics as described in the paper, this data must be collected at every training epoch. As with ActiVis, storing data for ten training epochs would take 350GB for a moderately sized network, creating a major bottleneck in using this technique. These intermediates could also be generated anew each time the analysis was to be run; however, to perform class sensitivity analysis, this would require running the model >200 times on the full dataset.

Network Dissection: Bau et. al. recently proposed Netdissect [4] to learn interpretable concepts for filters in a convolutional neural network (CNN). For every convolutional filter k, Netdissect calculates the distribution of values for the the activation maps $A_k(x)$ and computes a threshold T_k such that $p(A_k(x) > T_k) = \alpha$ where α is a small constant like 0.005. T_k is then used to binarize each A_k and the correlation between the binarized map and the original concept label is computed. Intermediates. Netdissect requires that the activation maps for every image and every convolutional unit be available. If Netdissect is to be run for a single unit or layer, it is conceivable that the computation can be done in memory. However, when performing this computation for all units or tuning the threshold T_k (e.g., for a new dataset), then it may be more efficient to store the intermediates vs. re-running the model repeatedly.

1.2 MISTIQUE: storing model intermediates

As demonstrated by the diagnostic techniques above, model intermediates form the substrate on which a variety of diagnostic and interpretability techniques are based. However, model intermediates require many tens to hundreds of GBs in storage, making it challenging to use existing diagnostic techniques as well as develop new ones. In addition, computing intermediates by re-running the model for each analytic query not only slows down the process of model diagnosis but can also be unacceptable for interactive query workloads. Thus, the bottleneck in supporting efficient and widely usable model diagnosis is caused by two data management questions: (a) how do we store large amounts of data efficiently for storage and querying (e.g., as in [6, 37, 48]); and (b) how do we trade-off intermediate storage vs. recreation (as in [7, 22, 52])?

To address these questions, we propose MISTIQUE, a system designed to capture, store, and query model intermediates for model diagnosis. MISTIQUE can work traditional ML pipelines as well as deep neural networks. MISTIQUE leverages unique properties of intermediates in both kinds of models to drastically reduce store costs while giving up little accuracy in most analytic techniques. Specifically, MISTIQUE is based on three key ideas: (1) Activation quantization and summarization: we take inspiration from existing diagnostic techniques to encode neuron activations based on data distributions, thus getting drastic storage reductions without trading off accuracy; (2) Similarity-based compression: we leverage data similarity in traditional ML pipelines as well as DNNs to remove redundancy between intermediates and obtain large compression ratios. (3) Adaptive querying and materialization: we propose a cost model to determine when a query for intermediates should be answered by re-running the model vs. reading a materialized intermediate. A similar cost model determines when an intermediate should be materialized. Together, these techniques can reduce storage for intermediates by up to 110X for traditional ML pipelines and 6X for deep neural networks, and provide a query speed-up of up to two orders of magnitude depending on the query. In all, we make the following contributions:

- We are the first to identify that model intermediates are a key substrate for a variety of model diagnosis and interpretation techniques, and that intermediate storage and querying is a bottleneck in implementing and using these techniques (Sec. 1).
- We present a set of diagnostic queries drawn from recent literature that represent commonly performed analyses on traditional ML pipelines and neural network models (Sec. 2.2).
- We propose MISTIQUE, a system to capture, store, and query intermediates for different types of ML models and pipelines. Our implementation supports pipelines built using scikit-learn as well as Tensorflow (Sec. 3).
- We propose three key optimizations to reduce storage footprint and speed up queries: (a) activation quantization and summarization (Sec. 4.1); (b) similarity-based compression (Sec. 4.2); and (c) adaptive querying and materialization (Sec. 4.3).
- We experimentally evaluate MISTIQUE on a set of pipelines in scikit-learn and neural network models in Tensorflow. We find that MISTIQUE reduces storage by up to 110X on traditional ML pipelines and up to 6X for deep neural networks. We show that our quantization strategies cause only a small reduction in diagnostic accuracy. Depending on the type of query, MISTIQUE provides a speedup of up to 390X for traditional ML pipelines and up to 210X for deep neural networks (Sec. 8).

2 PRELIMINARIES

In this section, we describe the models supported by MISTIQUE and our problem formulation. We also present a list of commonly used diagnostic techniques along with a categorization based on the amount of data used by each technique.

2.1 Models and Model Intermediates

In this work, we consider two classes of models: (a) *traditional* ML models built with hand-crafted features (denoted TRAD), and (2) deep neural network (denoted DNN) models that learn features from raw data. We work with a running example for each class of

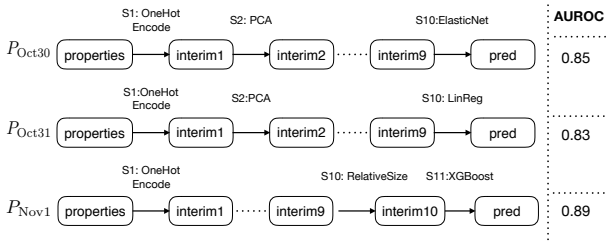


Figure 1: Zillow pipelines

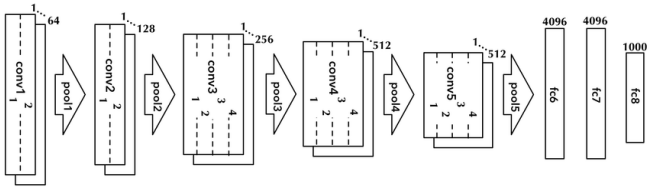


Figure 2: VGG16 architecture [46]

models. For TRAD, we consider the the set of models built to predict housing prices given attributes of the house such as number of rooms, area in square feet, and location (details about the task can be found in Sec. 7). Fig. 1 shows examples of TRAD models built for this task. Notice that these models are not a single monolithic entity; instead, they are comprised of a series of stages that make up the *model pipeline*. These stages are a mixture of data pre-processing steps (e.g., OneHotEncoding, Scaling), feature engineering steps (e.g., computing the RelativeSize feature), and model building and prediction stages (e.g., gradient boosted trees in XGBoost [14]).

For DNNs, we consider the set of models built for classifying images on the CIFAR10[1] dataset. This dataset consists of images drawn from ten classes such as frog, ship, deer, etc. Fig. 2 shows VGG16 [46][2], a popular DNN for image classification. Unlike TRAD, the DNN does not have explicit, separate stages for pre-processing, feature extraction, and prediction; instead, it has a large number of layers that play the role of feature extractors and a final layer that performs the prediction. Each layer in the DNN applies a different transformation to its input and represents the data in a different high-dimensional space. We refer to the operation applied at each stage (in the pipeline or DNN) as a *transformation* and the corresponding function or object as the *transformer* (e.g., OneHotEncoder, PCA, Convolutional Layer-5, SoftMax).

The process of training a model (TRAD or DNN) can produce different artifacts: the learned model parameters, log files, gradient information, intermediate datasets produced by different stages of the model, etc. In this paper, we focus on model intermediates that are the intermediate datasets produced by the different stages of the traditional ML pipeline or hidden representations (i.e., neuron activations produced by different layers in a neural network). For example, in Fig. 1, model intermediates are the outputs produced by every stage of the model pipeline (labeled "intermX"). Similarly, in Fig. 2, model intermediates are the data representations produced by every layer in the neural network. Thus, we find that while the structure of models in these two classes is distinct, intermediates produced by both classes of models are similar.

2.2 Characterization of Diagnostic Queries

In Sec. 1, we highlighted three techniques used for model diagnosis. In Table 1 we provide a survey of diagnostic and interpretability techniques drawn from the literature. For each diagnostic technique, we show an example of the question answered by that technique (or query) in terms of our running examples. For completeness, we also include analyses that cannot be handled solely with MISTIQUE either because they require access to data other than model intermediates (e.g., gradients) or because they require the ability to perturb data or models. Furthermore, to characterize the query performance of our system, we categorize diagnostic techniques based on amount of data required by each technique. Specifically, based on the number of *Rows*, i.e., input examples, and *Columns*, i.e., features, used by each technique, we define four categories: *Few Columns, Few Rows* (FCFR), *Few Columns, Many Rows* (FCMR), *Many Columns, Few Rows* (MCFR), and *Many Columns, Many Rows* (MCMR). In this work, *few* denotes <100. A typical diagnostic workload contains queries belonging to different categories; for example, one workload might be: (i) plot the prediction error for model P_{Nov1} (FCMR); (ii) for the house with highest prediction error, H^*, examine its raw features (MCFR); (iii) find the performance of houses P_{Nov1} on houses "most similar" to H^* (MCFR). (iv) plot the features of H^* vs. the average features of all houses (MCMR). A system for model diagnosis must therefore be able to support queries in all four categories. In Table 1, we identify by name (e.g., POINTQ) the queries that will be used in our experimental evaluation.

2.3 Problem Formulation

Each of the queries discussed above requires access to different intermediates from a model, e.g., predictions or hidden representations. For a given intermediate, there are two ways of computing it: (a) either we can re-run the model up to the particular intermediate (denoted RERUN) or (b) we can read the intermediate that has previously been materialized to disk (denoted READ). For instance, the Netdissect implementation from [4] re-runs the full model any time an analysis is to be performed. While this solution may suffice when computing intermediate results for a small number of examples, running the model over tens of thousands of examples is slow (e.g., up to two orders of magnitude slower than reading as shown in Sec. 8) and wastes computation. In contrast, systems like ActiVis [24] and VizML [13] store intermediates to disk and read them to answer queries. While materializing intermediates is essential for providing interactive query times, this can come at a large storage cost. As mentioned before, storing intermediates for ten epochs of the VGG16 network on CIFAR10 takes about 350GB (gzip compressed), a storage cost most developers are unwilling to pay. Similarly, storing fifty traditional ML pipelines with $9 - 19$ stages takes 67 GB of storage (gzip compressed). Thus, the strategies of RERUN and READ are optimal for some intermediates while they may be expensive (with respect to time or storage) for others. **In this work, we seek to address the question of speeding**

[1] https://www.cs.toronto.edu/ kriz/cifar.html
[2] in this work, we focus on image networks

Query Category	Specific instantiation	Intermediates Queried
	Queries Using Intermediates	
Few Columns, Few Rows (FCFR)	(POINTQ) Find the activation map for neuron-35 in layer-4 for image-345.png [53]	X, I
	(TOPK) Find the top-10 images that produce the highest activations for Neuron-35 in layer-13 [53]	X, I
	Get the predicted price error for Home-150 [3]	X, P
	Get accuracy of P_{Oct31} on the top-50 most expensive homes in LA [24]	Y, X, P
Few Columns, Many Rows (FCMR)	(COL_DIFF) Compare model performance for P_{Oct31} and P_{Nov1} grouped by type of house [31, 35]	X, Y, P
	(COL_DIST) Plot the error rates for all homes [13]	X, I, Y, P
	Find number of images that were predicted as a frog but were in fact a ship [13]	Y, P
	Compute the confusion matrix for the training dataset [3]	$\text{Y}_{\texttt{train}}$, $\text{P}_{\texttt{train}}$
Many Columns, Few Rows (MCFR)	(KNN) Find performance of `CIFAR10_CNN` for images similar to image-51 [3]	X, $x_{\text{img-51}}$, Y, P
	(ROW_DIFF) Compare features for Home-50 and Home-55 that are known to be in the same neighborhood but have very different prices [24]	I, Y
	Determine whether this test point is an adversarial example [17]	X, x_{test}, Y, y_{test}
	Find training examples that contributed to the prediction of this test example [26]	X, I, x_{test}, i_{test}
Many Columns, Many Rows (MCMR)	(SVCCA) Compute similarity between the logits of class ship and the representation learned by the last convolutional layer [39]	I
	(VIS) Plot the average activations for all neurons in layer-5 across all classes [24]	I
	Compare the representations learned in layer-5 by AlexNet and by VGG16 in Layer-8 [39]	$\text{I}_{\text{AlexNet}}$, I_{VGG}
	Find correlation between the activation of each neuron and pixels corresponding to concept lamps [4]	X, I
	Queries Not Using Intermediates	
Gradient-based	Find the salient pixels in Image-250 [43, 45, 57]	
Feature importance methods	Find importance of pixel-50 in this model [40, 44]	
Perturbing examples	Find the minimal change that must be made to mispredict Image-51 [18, 38]	
Training New Models	Find a smaller model that performs similarly to a larger model [20, 26]	

Table 1: A Categorization of Diagnostic Queries. Last column, X=input, Y=target, I=intermediate dataset, P=predictions.

up diagnostic queries by intelligently choosing when to re-run a model vs. (store and) read an intermediate and in turn minimizing the cost of storing intermediates.

3 MISTIQUE OVERVIEW

In this section, we give a high-level overview of the system architecture and how it can be used to run diagnostic queries.

3.1 Architecture

The system architecture for MISTIQUE is shown in Fig. 3. MISTIQUE consists of three primary components: the PipelineExecutor, the DataStore, and the ChunkReader. These three components are tied together by a central repository called the MetadataDB that is used to track metadata about intermediates and pipelines. The PipelineExecutor is responsible for running ML models and pipelines in logging mode. This means that the executor runs the pipeline forward, finds all intermediates produced by the pipeline and registers information about the model and intermediates in MetadataDB. The PipelineExecutor does not make decisions about data storage or placement; that falls under the purview of the DataStore. Along with logging intermediates, the PipelineExecutor is also responsible for storing transformers trained in a pipeline (e.g. a trained XGBoost model) so that the transformer may be re-run in the future without retraining.

Intermediates produced by the PipelineExecutor are passed on to the DataStore (Sec. 4) for decisions about whether and how to store the intermediate. The DataStore is made up of an InMemoryStore and a persistent store (on-disk in our implementation). MISTIQUE adopts a column-oriented scheme (much like [1, 48]) to store intermediates. Specifically, MISTIQUE represents each intermediate (including the source data and the final predictions) as a

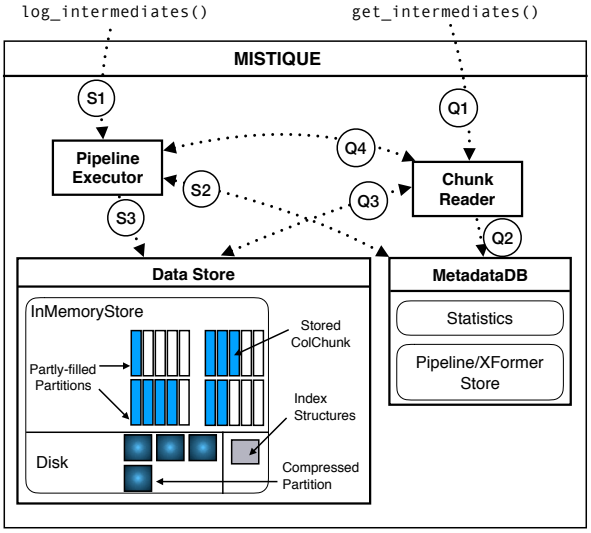


Figure 3: MISTIQUE **Architecture with data flow during storage (S1-3) and querying (Q1-4)**

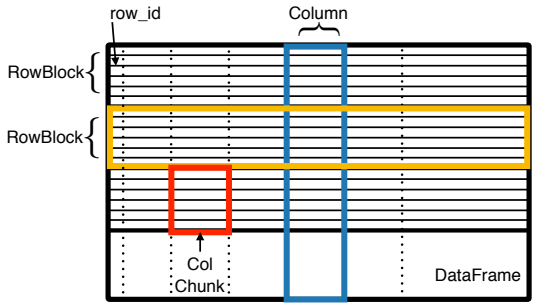


Figure 4: MISTIQUE **Data Model**

DataFrame[3] that is divided horizontally into a set of *RowBlocks*. Every row is associated with a unique `row_id` that is preserved across all intermediates and is used as a primary index. A DataFrame is also associated with a set of *Columns* that make up the DataFrame. The part of a Column that falls into a particular RowBlock is called a *ColumnChunk*. The data model is shown in Fig. 4.

The unit of data storage in the DataStore is a Partition. A Partition is a collection of ColumnChunks from *one or more* DataFrames that are to be stored together. The InMemoryStore serves as a bufferpool and keeps a number of (uncompressed) Partitions in memory. When a Partition is evicted from the InMemoryStore, the Partition is compressed and written out to disk. As described in subsequent sections, storage decisions in MISTIQUE are made at the level of ColumnChunks, giving the system fine-grained control over data placement. The DataStore also stores any indexes built on the data (by default, a primary index on `row_id`). The process of storing intermediates is indicated by S1—S3 in Fig. 3: the request to log intermediates is sent to the PipelineExecutor which sends each intermediate to the DataStore. The DataStore in turn queries the MetadataDB to determine whether the intermediate (or some columns) should be stored and, if so, stores the data using optimizations described below.

The final component of MISTIQUE is the ChunkReader (Sec. 6) which is responsible for servicing query requests. Query execution in MISTIQUE may involve fetching data from the DataStore or re-running pipelines to re-create intermediates. The procedure for querying intermediates is shown as Q1—Q4 in Fig. 3: the query is sent to the ChunkReader which queries the MetadataDB to determine whether to re-run the model or read data that was previously stored. Depending on the response, the ChunkReader either invokes the PipelineExecutor or queries the DataStore. The decision between these two alternatives is made by the cost model (Sec. 5). Regardless of how the data is obtained, a query to MISTIQUE produces a numpy array[4] that can be used as input to analytic functions.

3.2 Usage Example

To log intermediates from Tensorflow, the ML developer must only provide MISTIQUE with the path to the model checkpoint and as well as an input loading function. We currently support Tensorflow models built using Keras. Calling `log_intermediates(checkpoint, input_func)` causes MISTIQUE to run the model forward by calling the input function and log intermediates for every model layer. For scikit-learn pipelines, we have defined a YAML specification (modeled after Apache Airflow[5]) that is used to express scikit-learn pipelines in a standard format. We wrapped a number of common scikit-learn functions for use in the YAML specification (e.g. models like XGBoost, LinearRegression as well as preprocessing steps like Scaling and LabelIndexing). `log_intermediates(yaml_file, input_func)` similarly logs all the pipeline intermediates. Once intermediates have been logged in MISTIQUE, the ML developer can use a set of query APIs to access the data in MISTIQUE. The key query API exposed by MISTIQUE is `get_intermediates([keys])`. This API can be used to retrieve any column, of any intermediate, belonging to any model that has been logged with MISTIQUE. Keys take the form of `project.model.intermediate.column`. The API returns a numpy array with the required columns as well as the `row_id` column. For ease of use, MISTIQUE provides implementations of common analytic functions that can be applied on top of the numpy array result (although this is not the focus of our contribution). Since MISTIQUE returns numpy arrays, it is also easy to add other analytic functions.

4 DATA STORE

Once MISTIQUE has used the cost model (Sec. 5) to determine that an intermediate is to be stored, the DataStore is responsbile for determining how to most efficiently store the intermediate. The naïve strategy when storing intermediates is to fully store every itermediate from any pipeline that is run. While simple, this strategy requires a great deal of storage, e.g., it logs 350 GB of compressed data across ten epochs of the moderately sized `VGG16` model and requires 67 GB to store fifty traditional ML pipelines with <20 stages. Therefore, we explore different storage strategies to reduce footprint of intermediates without compromising query time or accuracy. Specifically, we propose three key optimizations: (a) for DNNs, we propose multiple quantization and summarization schemes to reduce the size of intermediates; (b) for DNNs as well as traditional pipelines, we perform exact and approximate deduplication between ColumnChunks within and across models; (c) we perform adaptive materialization of intermediates by trading off the increased storage cost with reduction in query time. We now expand upon each of these optimizations.

4.1 Quantization and Summarization

A key insight from diagnostic techniques proposed for DNNs is that ML developers are much more interested in *relative values* of neuron activations than they are in the exact values. For example, the visualizations in ActiVis are used to compare activations of neurons in different classes (see Fig. 12). Since the visualization cannot display >256 shades of the same color, at most 256 distinct activation values may be shown in the visualization. On similar lines, the Netdissect technique only examines neuron activations in the top 99.5th percentile, i.e., the technique only needs to know if the activation is "very high" or "not very high"—regardless of the actual activation value. This indicates that we can *quantize* or

[3]We choose to call it a dataframe because of the familiarity of the concept and not due to any parallels with R, pandas or Spark dataframes. They can equally be considered as relational tables.

[4]http://www.numpy.org/

[5]https://airflow.incubator.apache.org

discretize neuron activations into a much smaller number of values without affecting the accuracy of many diagnostic techniques. Previous work on model compression and model storage [19, 34] explored the use of quantization of *model weights* to reduce model size for inference as well as storage. In this work, we propose to extend those techniques to aggressively quantize neuron activations, noting that since these activations are only used for diagnostic purposes, we can perform drastic quantization. MISTIQUE supports three quantization schemes:

- *Lower precision float representation* (LP_QT): Storing a double precision float value as a single precision (float32) or half point precision (float16) value can lead respectively to a 2X and 4X reduction in storage with no effect on diagnostic accuracy.
- *k-bit quantization* (KBIT_QT): Since many diagnostic techniques are based on relative activations, we can reduce storage costs by representing values using quantiles (similar strategies are used to quantize weights in [34]). Given the maximum number of bits b to be allocated for storing each activation, we can compute 2^b bins using quantiles and assign each value to the corresponding bin. Quantization of activation values from o to b bits reduces storage by a factor of $\frac{o}{b}$.
- *Threshold-based quantization* (THRESHOLD_QT): To support queries such as Netdissect that use an explicit activation threshold, we can directly store data binarized using a given threshold. Once a threshold has been picked, however, we cannot binarize the data with respect to another threshold. This scheme reduces storage cost by o, the number of bits used by the original values.

The quantizations above reduce the storage required for each value, but do not reduce the *number* of values stored. This is particularly important in CNNs where the size of an activation map can significantly increase storage costs. Therefore, to reduce the number of activations, we support summarization via *pooling* (similar to the max-pooling operator in DNNs). In pooling quantization (POOL_QT), we apply an aggregation operation such as average (default) or max to adjacent cells in an activation map to obtain a lower resolution representation of the map. Assuming a 2-D activation map per channel (as in CNNs), pooling quantization reduces storage by $\frac{S^2}{\sigma^2}$ where we assume that the size of an activation map is SxS and size of the aggregation window is σxσ. We support two levels of pooling quantization: σ=2 (default, also denoted pool(2)) and σ=S, denoted pool(S), e.g., pool(32) for CIFAR10. $\sigma = S$ is the most extreme version of pool-based quantization where we compute a single average value to represent each activation map.

4.1.1 Implementation. Both KBIT_QT and THRESHOLD_QT require the system to first collect samples of activations to build a distribution and subsequently use this distribution to perform quantization. By default, for KBIT_QT, we set $k = 8$, which means that we compute $2^8 = 256$ quantiles for the activation distribution and assign each activation value to the appropriate quantile. 8BIT_QT reduces storage by 4X when raw activations are single precision floats and 8X for double precision. Note that when fetching an 8BIT_QT intermediate, we must also pay a *reconstruction cost* to go from the quantized values (0 - 255) to floating points. For POOL_QT, we conservatively use $\sigma = 2$. However, the user can choose to set a more aggressive pooling level depending on the application. In Sec. 8, we study the trade-offs involved in setting different σ values.

4.2 Exact and Approximate De-duplication

This optimization is based on two observations. First, intermediates in traditional ML pipelines often have many identical columns. For example, in the TRAD pipelines of Fig. 1, consecutive intermediates often only differ in a handful of features (e.g., RelativeSize between interm9 and interm10 in P_{Nov1}) and pipelines share many stages (e.g., in an extreme case like P_{Oct30} and P_{Oct31}, all intermediates are identical except for pred). Second, TRAD and DNN intermediates often have similar columns (e.g., predictions from multiple models for the same task such as P_{Oct30}, P_{Oct31}; intermediates from different epochs for the same DNN; and quantized versions of intermediates). We can leverage these insights to avoid storing redundant data and to compress similar data to obtain higher compression ratios.

4.2.1 Implementation. Implementing de-duplication (exact and approximate) requires two steps: first, we must identify identical or similar ColumnChunks, and second, we must compress these ColumnChunks when writing to storage (MISTIQUE does not currently compress ColumnChunks when in memory). We can identify identical ColumnChunks simply by computing the hash of the ColumnChunks. If an identical ColumnChunk has previously been stored, then the current ColumnChunk can be skipped. For detecting similar columns, we use the MinHash. For every new ColumnChunk, the DataStore computes the MinHash for the ColumnChunk (after discretizing the values) and queries the LSH index for Partitions with Jaccard similarity above a threshold τ. If an existing ColumnChunk is found to have similarity above τ, the new ColumnChunk is stored in the same partition as the existing ColumnChunk. Otherwise, a new Partition is created.

For DNNs, we perform two simplifications: (a) we only perform exact de-duplication because DNN columns seldom have similar values; (b) we co-locate columns from the same intermediate because DNN intermediates (when flattened) have many more columns than TRAD intermediates and therefore data similarity based co-location disperses columns over a large number of Partitions.

The previous procedure performs a rough clustering of ColumnChunks based on similarity and assigns ColumnChunks to Partitions. When a Partition is to be written to disk (e.g., because it is full or gets evicted from the InMemoryStore), the Partition is compressed and written out. MISTIQUE supports a variety of off-the-shelf compression schemes including gzip, HDF5, and Parquet.

4.3 Adaptive Materialization

Adaptive materialization is motivated by the observation that traditional ML pipelines and DNN models are often many stages long but not all intermediates or columns are accessed with equal frequency. Some intermediates (e.g., predictions of a model or image embeddings from the last convolutional block) may be accessed very often while others (e.g., activations from the first convolutional layer) may be accessed less frequently. Therefore, we trade-off the increase in storage cost due to materialization against the resulting speedup in query time. We capture this trade-off in parameter γ formally described in our storage cost model (Sec. 5.2). If γ for an intermediate is larger than a threshold, the intermediate is materialized. An intermediate that is queried a large number of times has a large γ value and is more likely to be materialized. Similarly, an intermediate with a small storage cost leads to a larger γ and

is more likely to be materialized. The full algorithm for logging intermediates is shown in Alg. 4 in the Appendix.

5 COST MODEL

In order to make the decision about (a) whether to store an intermediate, and (b) whether to execute a query by re-running a model or reading an intermediate, we respectively develop a storage cost model and a query cost model. We begin with the query cost model which provides the building blocks for the storage cost model.

5.1 Query Cost Model

The total time to execute a diagnostic technique (t_{diag}) can be computed as the sum of the time to fetch the required intermediates (t_{fetch}) and the time to perform computation on the data (t_{compute}).

$$t_{\text{diag}} = t_{\text{fetch}} + t_{\text{compute}} \tag{1}$$

The time to fetch the data, in turn, is equal to either the time to re-run the model or to read a previously stored intermediate. Since the compute time is the same in both cases, we only model t_{fetch}. Suppose we want to fetch the intermediate at stage i in model M (e.g., i-th layer in a DNN or stage-i in a traditional ML pipeline) and we seek to compute the intermediate for n_ex examples (where n_ex is between 1 and TOTAL_EXAMPLES). Then, if $t_{\text{i,re-run}}$ denotes the time to re-run the model to intermediate i, we can compute this quantity as the sum over each stage s $\leq$ i of: (a) time to read the transformer for s ($t_{\text{read\_xformer}}$), (b) time to load the input for s ($t_{\text{read\_xformer\_input}}$), and (c) time to execute s ($t_{\text{exec\_xformer}}$).

$$\begin{aligned} t_{\text{i,re-run}} = \Sigma_{s=0}^{i} \{ & t_{\text{read\_xformer}}(s) + t_{\text{read\_xformer\_input}}(s) \\ & + t_{\text{exec\_xformer}}(s) \} \end{aligned} \tag{2}$$

For a DNN, we can rewrite the model as follows: (a) we usually load the entire model at once, so we can rewrite the first term as $t_{\text{model\_load}}$; (b) explicit input to the DNN is only provided at layer-0, so we read input once; and (c) since prediction usually occurs in batches, we factor batch size into the time to execute a stage. If $t_{\text{model\_load}}$ denotes the time to read the model, sizeof denotes the size of an object in bytes, bt_size denotes the batch size, ρ denotes the read speed of storage, and t_{fwd}(s, bt_size) denotes the time to run a single batch of examples through the DNN up to layer s, the cost model for re-running DNNs can be written as in Eq. 3.

$$\begin{aligned} t_{\text{i,re-run,NN}} = t_{\text{model\_load}} + \frac{\text{n\_ex} \cdot \text{sizeof(ex)}}{\rho} \\ + \frac{\text{n\_ex}}{\text{bt\_size}} \cdot \Sigma_{s=0}^{i} t_{\text{fwd}}(s, \text{bt\_size}) \end{aligned} \tag{3}$$

$$t_{\text{i,read}} = \frac{\text{n\_ex} \cdot \text{sizeof(ex)}}{\rho_d} \tag{4}$$

Instead of re-running a model to obtain an intermediate, we can also read a previously-stored intermediate. The time to read an intermediate (denoted t_{read}) is simply proportional to the size of the intermediate, i.e., the number of examples multiplied by the size of one example. We assume that the size of the example accounts for the precision of the value (e.g., float16, uint8). We fold the time to read, decompress, and reconstruct the data into the constant ρ_d.

If $t_{re\text{-}run} \geq t_{read}$, we run the query by reading a previously stored intermediate.

5.2 Storage Cost Model

Using the query cost model, we can also determine when to store (i.e., materialize) an intermediate. The decision to materialize can be made at the level of an entire intermediate (i) or a particular column in an intermediate (i, c). In either case, we compute $t_{\text{re-run}}$ and t_{read} as above by setting n_ex to TOTAL_EXAMPLES. If $t_{\text{re-run}} \geq t_{\text{read}}$, we can trade-off the speedup from storing the intermediate against the additional cost of storing the intermediate.

$$\gamma = \frac{(t_{\text{i,re-run}} - t_{\text{i,read}}) \cdot \text{n\_query(i)}}{\text{S(i)}} \tag{5}$$

This trade-off is captured in γ shown in Eq. 5 where S(i) is the (amortized) storage required for intermediate i, n_query(i) is the number of queries made to intermediate i that gets updated with each query to the system. The units of γ *are seconds per GB* and it captures the amount of query time saved per GB of data stored. For example, a γ of 1000 sec/GB indicates that the ML developer is willing to use 1GB of storage in exchange for a total saving of 1000s in query time. Note that the numerator of Eq. 5 increases with the number of times intermediate i is queried. S(i), in turn, is affected by storage of other intermediates; intermediates with similar data will lead to lower S(i) because they are compressed together.

6 FETCHING DATA FROM MISTIQUE

Diagnostic techniques like those discussed in Sec. 2.2 are executed by first fetching the data from MISTIQUE and then running analyses on it. We currently have a simple query execution model inside MISTIQUE. The ChunkReader is responsible for fetching intermediates either by reading them from the DataStore or re-running the pipeline and returning them to the user. When a query for an intermediate arrives, the ChunkReader first queries the MetadataDB to check if the intermediate has been stored and if so, verifies that the time to read the intermediate is less than the time to re-run the model (as computed by the cost model in Sec. 5). If the time to read is smaller, the ChunkReader queries the DataStore for the intermediate. The DataStore in turn identifies the Partitions containing ColumnChunks for this intermediate. For particular kinds of queries (e.g., fetch results by row_id), MISTIQUE can use the primary index to speed up retrieval of relevant Partitions. In the future, we can incorporate specialized indexes for particular types of queries (e.g., nearest neighbor index). Once the relevant Partitions have been read, the ColumnChunks for this intermediate are stitched together and returned. On the other hand, if it is faster to re-run the model, the ChunkReader invokes the PipelineExecutor to obtain the intermediate. The PipelineExecutor in turn executes previously stored transformers for this model or pipeline. In either case, the result of the query to MISTIQUE is a numpy array which is then used as input for further analysis (e.g., SVCCA, visualization). Pseudocode for querying MISTIQUE is shown in Alg. 3 in the Appendix.

7 EXPERIMENTAL SETUP

In the previous sections, we described the data model, architecture and optimizations implemented in MISTIQUE. In this section, we present an experimental evaluation of our system on multiple

real-world models and analytical techniques. We begin with a description of the experiment setup, including the generation of ML pipelines, and then describe our results.

7.1 Workflows

We evaluated our storage techniques on different traditional ML pipelines from scikit-learn and on deep neural network models built in Tensorflow.

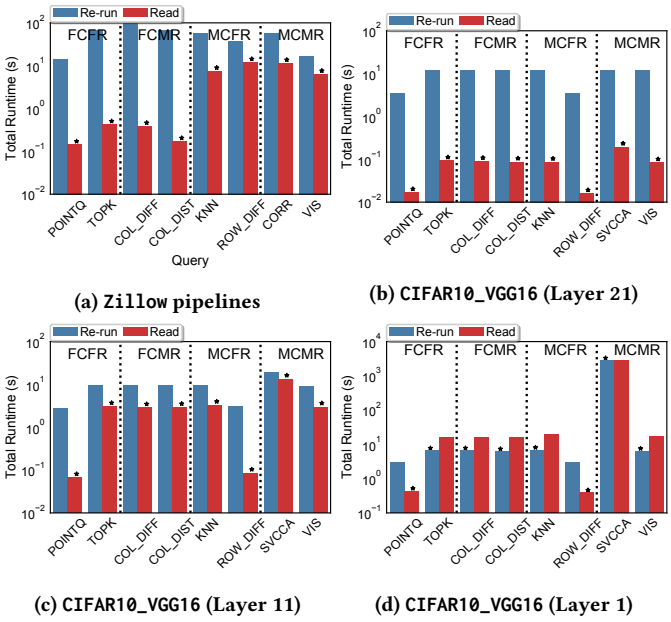
7.1.1 Traditional ML Models. (TRAD) To replicate a real-world machine learning scenario for traditional models, we took the dataset and task from the Kaggle Zestimate competition[6]. The goal of this competition is to use data about homes (e.g., number of rooms, average square footage) to build a model that can predict the *error* of Zillow's in-house price prediction model. To obtain pipelines for this task, we took scikit-learn scripts uploaded by Kagglers for the Zestimate competition and turned them into workflows in MISTIQUE. The workflows contained between 9 - 19 different stages including data preprocessing, feature engineering, feature selection, fitting a model, and making predictions using the model. Since competitors often submit scripts with the best hyperparameter settings, we also defined pipeline variations by changing the hyperparameter settings. The result of this process was a set of 50 pipelines that are described fully in Appendix E. We note that in real modeling projects, data scientists build many more than 50 models; however, the gains offered by MISTIQUE only grow with more pipelines and therefore, we chose to limit our experimentation to 50.

7.1.2 DNN Models. (DNN) To illustrate the efficacy of our techniques on deep neural networks, we work with the CIFAR10 image classification dataset. CIFAR10 contains 50K training images from 10 classes where each image has dimensions 64x64x3. We evaluate on two models trained on CIFAR10: the VGG16 model fine-tuned on CIFAR10, denoted as CIFAR10_VGG16 (the original model has been trained on the IMAGENET [41] dataset) and a well-accepted, simple CNN model trained from scratch, denoted as CIFAR10_CNN [7]. The original VGG16 model consists of 13 convolutional layers and 3 fully connected layers. During fine-tuning of VGG16, we take the first 13 pre-trained convolutional layers, freeze their weights, replace the original fully connected layers with two smaller, fully connected layers (because CIFAR10 does not require these layers to be wide) and train these layers. In contrast, CIFAR10_CNN has 4 convolutional layers and 2 fully connected layers, and is trained from scratch. We checkpoint model weights after every 10% of the total number of epochs (i.e., 10 checkpoints each).

MISTIQUE has been entirely implemented in Python. All experiments were run on an Intel Core i7-6900K machine running at 3.20 Ghz. 32 core machine (16 CPUs) with 64 GB RAM, and 2 GM200 GeForce GTX Titan X GPU. GPU support was enabled when running DNN models.

8 EXPERIMENTAL RESULTS

Our goals in the experimental evaluation are to answer the following key questions: (1) What is the speedup in execution time from using MISTIQUE to run diagnostic queries? (2) What overall storage gains can we achieve by using MISTIQUE and our proposed optimizations? (3) Does our cost model accurately capture the re-run vs. read trade-off? (4) For DNNs, how do the proposed quantization schemes affect accuracy of diagnostic techniques? (5) What is the overhead of using MISTIQUE vs. baselines? (6) What is the impact of adaptive materialization on storage and query time?



(a) Zillow pipelines
(b) CIFAR10_VGG16 (Layer 21)
(c) CIFAR10_VGG16 (Layer 11)
(d) CIFAR10_VGG16 (Layer 1)

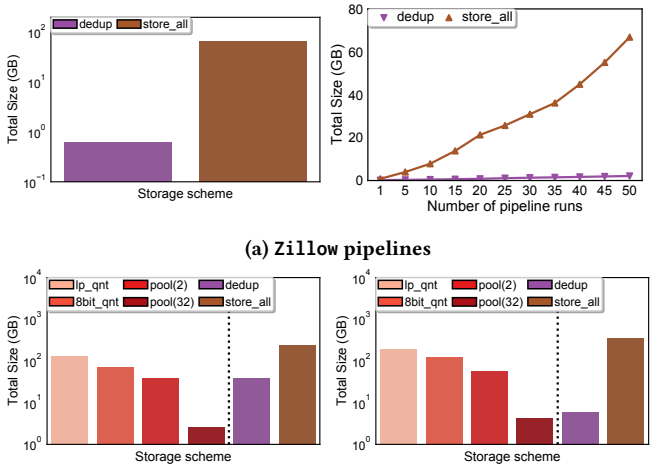
Figure 5: End-to-end query runtimes. Asterisk indicates strategy picked by cost model

8.1 End-to-End Query Execution Times

In this set of experiments, we evaluate the end-to-end execution times for a representative set of diagnostic techniques from Table. 1. For each query category (FCFR, FCMR, MCFR, MCMR), we evaluate on two queries for the TRAD and DNN models (see Table. 5 in the Appendix for full list). For the DNN queries, we run the same query at multiple layers to show how the trade-off between reading and re-running changes across layers. In this experimental setup, when re-running DNN models, we pre-fetch the entire input into memory. Batch size for the DNN queries was set to 1000 and RowBlocks in MISTIQUE were set to be 1K rows. Fig. 5 shows results of this experiment with an *asterisk indicating the strategy chosen by the cost model*. Note the log scale on the y-axes.

For TRAD models, we find that running a query by reading an intermediate is always faster than re-running the pipeline. For example, in Fig. 5a, we see a speedup of between 2.5X — 390X. In contrast, for DNN models (here CIFAR10_VGG16), the decision between whether to re-run or read depends on the model layer and number of examples fetched by the query. For queries on Layer21 (Fig.5b), the last layer, reading intermediates is 60X — 210X faster that than re-running the model. For Layer11 (Fig.5c), we see that reading the intermediate is again faster by 2X — 42X. In contrast, we find that at Layer1 (Fig.5d), *re-running* the model can be up to 2.5X faster for some queries. This is because Layer1 is very

[6] https://www.kaggle.com/c/zillow-prize-1
[7] https://github.com/keras-team/keras/blob/master/examples/cifar10_cnn.py



(a) `Zillow` pipelines

(b) `CIFAR10_CNN` (left) and `CIFAR10_VGG16` (right)

Figure 6: Storage sizes for different strategies

close to the input and is the largest layer by size (see Sec. D in the Appendix). Since we pre-fetched input data for DNN queries, we expect to observe even larger speedups when the input must be read from disk. For all queries we evaluated, with the exception of SVCCA, compute time is a small fraction of the total query time. For SVCCA on Layer1, in contrast, compute time accounts for about 99% of the total time.

The above experiment empirically demonstrates that choosing the right strategy (re-run vs. read) in executing a diagnostic technique can lead to speedups of between 2X— 390X.

8.2 Intermediate Storage Cost

Next we examine the storage gains obtained by the different optimizations proposed in MISTIQUE. Fig. 6 shows the storage cost, i.e., number of bytes used, for `Zillow`, `CIFAR10_CNN` and `CIFAR10_VGG16` models. For every set of models, we store intermediates from *all* stages. For TRAD, we evaluate the basic strategies of STORE_ALL and DEDUP. For DNN models, we also compare the storage cost when applying different quantization schemes: LP_QT, 8BIT_QT, and POOL_QT ($\sigma = 2, 32$) described in Sec. 4.

For `Zillow` (Fig. 6a, left), we find that the raw dataset is 168 MB compressed but STORE_ALL requires 67GB to store all the intermediates across 50 pipelines. The 400X larger storage footprint indicates that the naïve STORE_ALL strategy cannot scale to large input data, long pipelines or large number of models. In contrast, DEDUP, which applies approximate and exact de-duplication as discussed in Sec. 4.2, drastically reduces storage cost by 110X to 611 MB. On the right side of Fig. 6a, we see that the cumulative storage cost for `Zillow` increases linearly for STORE_ALL, while it stays relatively constant for the DEDUP strategy. This is because, for `Zillow` (and most TRAD pipelines), many columns are shared between pipelines. Consequently, most of the storage cost is due to the first pipeline whereas deltas are stored for the rest.

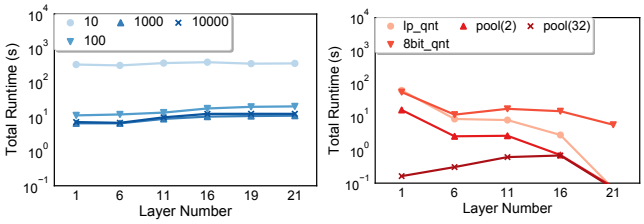
In Fig. 6b we show the cost in bytes of storing intermediates for `CIFAR10_CNN` and `CIFAR10_VGG16`. For both models, we store intermediates for ten epochs. The raw size of the input data (`CIFAR10`) is 170MB (compressed) in both cases. For `CIFAR10_CNN` we find that STORE_ALL with no quantization requires 242 GB, while STORE_ALL consumes 350 GB for `CIFAR10_VGG16`. The storage cost per DNN model is much larger than that of a traditional ML pipeline. As a result, reducing storage footprint is even more essential for DNN models. For each model, we present storage sizes for LP_QT, 8BIT_QT and POOL_QT (σ=2,32). We apply DEDUP on top of the default scheme of POOL_QT (σ=2) to obtain the final size.

For `CIFAR10_CNN` , we see that LP_QT reduces storage from 242GB to 128 GB for and 8BIT_QT further reduces it to 72.4 GB. The biggest storage gains can be achieved by applying different levels of POOL_QT which can reduce storage to 39 GB for $\sigma = 2$ (6.2X reduction) and 2.53 GB for $\sigma = 32$ (95X reduction). Applying DEDUP does not produce significant gains because, unlike `Zillow` pipelines, `CIFAR10_CNN` has few or no repeated columns. `CIFAR10_VGG16` shows the same trends as that of `CIFAR10_CNN` except for the impact of DEDUP. Applying POOL_QT to `CIFAR10_VGG16` reduces storage by 6X for $\sigma = 2$ (58 GB) and by 83X for $\sigma = 32$ (4.19 GB). As discussed in Sec. 7, `CIFAR10_VGG16` is trained such that the bottom 13 convolutional layers of the network are frozen while only the top fully connected layers are trained. Thus, intermediates from the 13 layers are the same across all models and therefore applying DEDUP in addition to POOL_QT ($\sigma = 2$) reduces storage footprint by 60X to 5.997 GB.

Thus, MISTIQUE **can reduce storage footprint by between 6X (DNN)— 110X (TRAD), and upto 60X for fine-tuned models.**

8.3 Validating the Cost Model

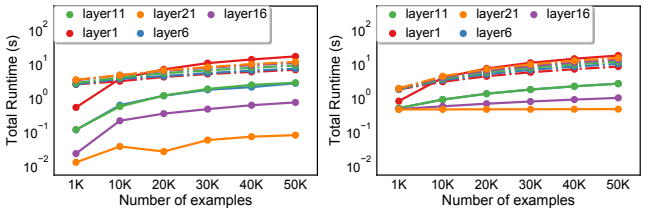
In Sec. 5, we proposed models to quantify the cost of re-running a model and the cost of reading an intermediate. In this section, we present experiments verifying our cost models and the resulting trade-off, focusing particularly on DNNs. For this evaluation, we used the `CIFAR10_VGG16` model and stored intermediates for all layers to disk. Next, we ran an experiment where we fetched each intermediate either by reading the intermediate from disk or by re-running the model. When re-running the DNN model, we pre-fetched the entire input data into memory to avoid disk access. We repeated this process for different number of examples (n_ex in the cost model). The results are shown in Fig. 7 and 8.



(a) Cost of re-running model **(b) Cost of reading intermediate**

Figure 7: Verifying the cost model

Fig. 7a shows the time required to compute intermediates by re-running the model (n_ex=TOTAL_EXAMPLES) with different batch size settings. We see that batch size has a significant impact on execution time (since it affects the number of times the model is run forward). Computing intermediates for batch size=10 is 30X slower than for batch size=1000. Performance degrades slightly for batch size of 10000, whereas larger batchsizes overflow the GPU memory



(a) Empirical (b) Predicted

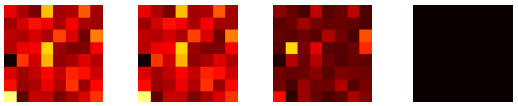
Figure 8: Read (solid) vs. Exec (dashed) Trade-off

(11 GB). (We therefore use batch size of 1000 in all experiments.) We also find that time to re-run increases proportionately to the layer number and we pay a fixed cost of 1.2s for model load. As shown in Fig. 8a, execution time also increases linearly with n_ex.

Fig. 7b shows the time to read an intermediate from disk for different layers and quantization schemes. The RowBlock size was set to 1K rows and n_ex=TOTAL_EXAMPLES. As captured in Eq. 4, the time to read an intermediate depends on the number of examples in the intermediate and size of each example. We find that `8BIT_QT` has the largest read time (due to high reconstruction cost), followed by `LP_QT` (but with 2X as much storage as `8BIT_QT`), followed by pool(2), and finishing with pool(32). Although pool(32) produces the best query time, the drastic summarization makes it impossible to run certain queries on it (e.g., SVCCA). **pool(2) therefore presents a good trade-off with respect to query time and storage, and we use it as the default storage scheme in** MISTIQUE**.**

Next, we examine the impact of the cost difference between re-running and reading an intermediate when querying different layers of the `CIFAR10_VGG16` model. When reading intermediates, we assume that the intermediates have been stored with the default pool(2) scheme. Fig. 8a shows the retrieval cost for five different layers when n_ex is varied between 1K — 50K. Dashed lines correspond to re-running the model whereas solid lines correspond to reading the intermediate. We find that (as captured in our cost model), the time to read or re-run intermediates scales linearly with the number of examples. Similar to the trend in end-to-end runtimes, reading intermediates is cheaper than re-running the model for all layers except Layer1 (>10K examples). Layer1 is anomalous because the intermediate is of large size (and therefore takes long to read) but is close to the input (and therefore is fast to re-run). Fig. 8b shows the same trade-off from Fig. 8a, except as predicted by our cost model. We see that the cost model accurately predicts the trade-off between re-running vs. reading and can be used to determine the right query execution strategy (as in Fig. 5.

One constraint when querying an intermediate via reading is that the number of RowBlocks read depends on the whether the examples queried are scattered and whether there is an appropriate index available on the RowBlocks. However, since the dotted and solid lines in Fig. 8 do not intersect, we see that even if MISTIQUE has to read the entire intermediate (50K) examples, it is faster to read the intermediate vs. re-run the model. In addition, while `RERUN` can only benefit from indexes on the input (e.g., find predictions for examples 36), MISTIQUE can index any intermediate and speed up queries in different layers (e.g., find predictions for examples with neuron-50 activation > 0.5).



(a) (b) (c) (d)

Figure 9: Visualizing average activations for different storage schemes: (a) full precision (float32), (b) `LP_QT` (float16), `KBIT_QT` (k = 8), `POOL_QT` (σ = 32) (all produce the same heatmap); (c) `KBIT_QT` (k = 3); (d) `THRESHOLD_QT`

8.4 Effect of Quantization on Accuracy

Next, we discuss the effect of our quantization strategies on diagnostic techniques. We highlight results from three queries, namely, VIS, SVCCA and KNN from Table 1. (1) VIS: Similar to [24], suppose we want to visualize the average activation of 256 neurons in layer-9 of the `CIFAR10_VGG16` network. Fig. 9 shows heatmaps of these activations for full precision values (float32), `LP_QT` (float16), `KBIT_QT` (k=8), `POOL_QT` (σ = 32), `KBIT_QT` (k=3) and `THRESHOLD_QT` (99.5%). We see that there is no visual difference between full precision, `LP_QT` (float16), `KBIT_QT` (k=8) and `POOL_QT` (σ=32 or equivalently σ=2). However, `KBIT_QT` (k=3) and `THRESHOLD_QT` show obvious visual discrepancies. (2) SVCCA: The results of performing CCA [39] (captured in the average cca coefficient) between the logits produced by the `CIFAR10_VGG16` network and representations of four different layers are shown in Table 2. We see that the cca coefficient for the `8BIT_QT` intermediate is extremely similar to the full precision intermediate. In contrast, `POOL_QT` (σ = 2) introduces a discrepancy in the coefficient that reduces as the layer number increases. While `8BIT_QT` is more accurate, reading `8BIT_QT` is 6X slower and takes 1.5X more storage than pool(2). (3) K-nearest neighbors (KNN): In KNN, our goal is to find the k most similar examples to a given example similar to [3]. Table 3 in the Appendix shows the accuracy of KNN on different layers when using `8BIT_QT` and `POOL_QT` (σ = 2). Here, we set k=50 and measure accuracy as the fraction of nearest neighbors that overlap with the true nearest neighbors computed on the full precision data. As with SVCCA, we find that `8BIT_QT` produces almost the same neighbors as the full precision intermediates whereas `POOL_QT` usually captures 75% of the neighbors.

Thus, we find that `8BIT_QT` is more accurate than pool(2) for some diagnostic queries; however, the increased accuracy comes at the cost of 1.5X more storage and 6X slower queries. In MISTIQUE, we choose to accept this lower accuracy of pool(2) but provide the user the option of using `8BIT_QT` as the default storage scheme.

Layer	Full precision	8BIT_QT	pool(2)
SVCCA (value of average cca coefficient)			
6	0.8886	0.8868	0.6098
11	0.9185	0.9176	0.7085
16	0.7891	0.787	0.7464
19	0.8182	0.8182	0.8086

Table 2: SVCCA accuracy: Comparison of CCA coefficient across different storage schemes

8.5 Adaptive Materialization

In Sec. 5.2, we proposed a simple cost model to trade-off storage for an intermediate vs. the resulting decrease in query time. The

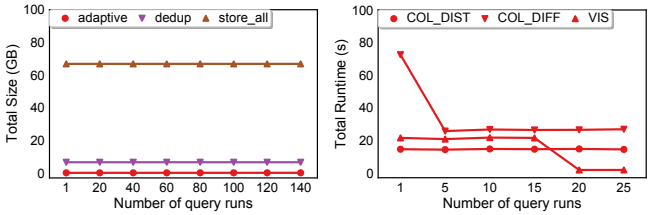


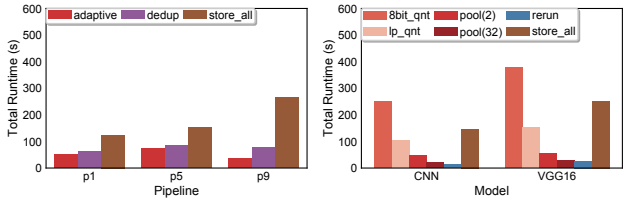
Figure 10: Adaptive Materialization: effect on storage and query time for synthetic `Zillow` workload

impact of adaptive materialization is highly workload dependent. In this evaluation, we demonstrate the efficacy of this optimization on a synthetic workload. We generated the synthetic workload by randomly choosing 25 queries (with repetition) for `Zillow` from Table 5. We then used MISTIQUE to log intermediates with adaptive materialization turned on. We set γ to 0.5s/KB (i.e., trade-off 1 KB of storage for a 0.5s speedup in query time). Fig. 10 shows the impact of this optimization on storage size and runtime of queries. On the left, we see that adaptive materialization (`ADAPTIVE`) has an extremely small storage footprint compared to both `STORE_ALL` and `DEDUP`: intermediates are materialized only once an intermediate has been queried a large number of times. On the right of Fig. 10 we see the query times for three different queries (chosen to demonstrate three different behaviors). When no columns have been materialized, queries in the adaptive strategy take as long as `RERUN`. As more queries are executed and columns are materialized, the response time for queries reduces. For example, the response time for VIS query reduces from 20s to 1.7s after 15 queries whereas that of COL_DIFF reduces from 75s to 26s after 5 queries. The response time for COL_DIST however remains unchanged. Thus, with the appropriate γ value, adaptive materialization can find a good trade-off between increased storage and decreased query time.

8.6 Pipeline Overhead

As shown in the architecture diagram in Fig. 3, a new intermediate that is to be logged in MISTIQUE is first added to the InMemoryStore. Partitions from the InMemoryStore are written to disk only if the Partition is full or the Partition gets evicted from the InMemoryStore. Therefore, the exact overhead of logging depends on whether the the relevant Partitions are full and if the InMemoryStore is already saturated: if the InMemoryStore is saturated, then logging an intermediate will result in a write to disk; however, if the InMemoryStore is not saturated, then there is no overhead associated with logging. Since InMemoryStore and Partition saturation depend closely on the workload, it is challenging to accurately estimate logging overhead in general. Instead, to provide an **upperbound** on logging overhead, we measure pipeline execution time when each intermediate is written to disk synchronously.

Fig. 11 shows the total runtimes (including logging overhead) for three `TRAD` pipelines, P1, P5 and P9 (see Table. 4 in the Appendix for full pipeline definitions). These were picked as representative pipelines because of varying lengths and use of diverse models (they contain 12, 17, and 18 stages respectively). We find that pipeline runtime is directly correlated with the amount of data written to storage. The `STORE_ALL` strategy consistently produces the largest pipeline execution time since it writes the largest amount of data



(a) `Zillow` pipelines **(b) `DNN` models**

Figure 11: Logging Overhead

(see Fig. 6). `ADAPTIVE`, in contrast, has low but non-zero overhead (because it stores transformations used in the pipeline). The `DEDUP` strategy produces modest overhead that is close to `ADAPTIVE` because it stores little data over each pipeline.

For `DNN`s, we find that running the `CIFAR10_VGG16` model without logging takes 19s. Storing all intermediates (without compression) takes 252s for single precision floats and 151s for half precision floats. When using `8BIT_QT`, we pay an extra cost for computing quantiles and binning the data, resulting in pipeline execution time of 379s. While `8BIT_QT` takes 13X longer than running the model with no logging, this overhead is small compared to the time taken to train a model (often >30 mins for this model). Finally, using `POOL_QT` ($\sigma = 32$) on `CIFAR10_VGG16` results in execution time of 20s — comparable to the time to run the model while $\sigma = 2$ requires 56s and $\sigma = 4$ requires 38s. Since `CIFAR10_CNN` shows similar trends, we do not discuss its logging overhead separately.

9 RELATED WORK

Model lifecycle management. Our work is most similar to previously proposed systems for lifecycle management of ML models (e.g., [27, 34, 47, 50]). ModelDB [50] was the first to introduce the problem of managing different model versions and storing metadata about models to support reproducibility. It did not, however, focus on storing intermediate datasets and we imagine that MISTIQUE could be integrated into a solution like ModelDB. ModelHub [34], in contrast, focused on the problem of storing model weights (also multi-dimensional arrays, like model intermediates) across different models and versions. As a result, ModelHub and MISTIQUE share similar goals and some of proposed techniques (e.g., quantization) also have similarities. ModelHub does not, however, address the problem of storing model intermediates or supporting model diagnosis. In addition, we note that the proposed algorithms for optimally determining which model versions to materialize have limited applicability in our setting since the set of model intermediates we wish to store is not known ahead of time. [47] proposed KeystoneML, a framework to express ML pipelines using high-level primitives and optimize their execution in a distributed setting.

Model diagnosis. As discussed in the introduction and referenced in Table 1, many techniques have recently been proposed for model diagnosis and interpretability. These include visualization tools such as ActiVis [24], VizML [13], ModelTracker [3] and DeepVis Toolbox [53] that allow users to inspect data representations learned by the model. Other diagnostic techniques propose to use surrogate models to explain model behavior and failures (e.g., PALM [26], LIME [40], [2, 20]). Yet other techniques have been proposed that perform model diagnosis using model gradients and backpropagation (e.g., [43, 45, 53, 55]) or data perturbation (e.g., [25, 38, 44]).

Another class of techniques such as Netdissect [4] and SVCCA [39] examine hidden representations of models to answer questions such as 'what real-world concepts are encoded in each neuron' and 'whether the representations learned by two models are the same'. A large portion of diagnostic and interpretability techniques described above work on model intermediates and can therefore benefit from a system like MISTIQUE.

Provenance capture and storage. Lineage or provenance capture and storage has been a rich area of work in the database community (e.g., surveys [9, 10]). Many systems have been proposed to capture coarse-grained lineage information, i.e., the set of transformations applied to a dataset. These include scientific workflow systems (e.g., VisTrails [11], Taverna [51], Kepler [28]), ML lifecycle management systems (e.g. ModelDB and ProvDB [33]) and also distributed processing systems (e.g., RDDs in Apache Spark [54]). Similarly, many systems have been proposed to collect and query fine-grained lineage data (example-level or cell-level) for specific data types and computation models. Some examples include SubZero [52] for array-data, Titian [23] for Apache Spark, Trio [5], Panda [22] for relational data and data warehouses, and [56] for ML pipelines in KeystoneML.

As a result, some of the questions addressed in this paper, have been explored in previous work. The key differences between MISTIQUE and work on lineage are three-fold: First, MISTIQUE already has coarse-grained lineage available through the YAML specification or the model checkpoint, so it needs to perform no extra work here. Second, the questions answered by lineage systems are significantly different from those answered by a model diagnosis system. For instance, a lineage system seeks to answer queries of the form "what input record produced a particular prediction?" (which, incidentally, can be answered easily based on row_id) whereas MISTIQUE seeks to answer queries such as "find all the input examples that had high value for a given feature." And third, in many ML models, we do not require specialized systems for fine-grained (cell-level) lineage since this data can be obtained via the existing forward and backward propagation mechanisms (e.g., in DNNs).

Data model, versioning, and compression. In this work, we take a "relational" view of model intermediates and represent data in a columnar format similar to [12, 48]. We could alternatively treat intermediates as multi-dimensional arrays and use array-based storage systems such as TileDB [37] and SciDB [49]. Since our goal is to efficiently store intermediate datasets, our work is related to work on dataset versioning and storage in both relational as well as array-systems (e.g., [6–8, 21, 30, 42]). For example, the DataHub [6] system proposed an architecture to perform collaborative data analysis by sharing datasets, Decibel [30] proposed efficient techniques to store relational dataset versions, and [21] proposed "bolt-on" versioning for traditional relational databases. On the side of array databases, [42] tackled the question of storing multiple versions of array data by taking advantage of delta encoding and compression. Subsequent work [7] in a similar vein, but for relational data, addressed the question of storing vs. re-creating dataset versions. While the techniques proposed in this work are powerful, they have limited applicability in our setting because our intermediates are not versions of the same dataset and the complete set of versions is not known apriori. Besides dataset versioning, our proposed quantization strategies are similar to those used to compress neural network weights in [19] and scientific data [8]. Since MISTIQUE stores data in a compressed format, in the future, we could also incorporate analysis techniques operating directly on compressed data as in [16]. Our data de-duplication strategy is related to de-duplication techniques used in block-oriented storage systems [32].

10 CONCLUSION AND FUTURE WORK

Model diagnosis is an essential part of the model building process. Analyses performed during model diagnosis often require access to model intermediates such as features generated via feature engineering and embeddings learned by deep neural networks. Querying these intermediates for diagnosis requires either the intermediate to have been pre-computed and stored or to be re-created on the fly. As we demonstrate in this paper, making an incorrect decision regarding reading vs. re-running can slow down diagnostic techniques by up to two orders of magnitude. In this work, we proposed a system called MISTIQUE tailored to capture, store, and query model intermediates generated from machine learning models. MISTIQUE uses a cost model to determine when to re-run a model vs. read an intermediate from storage. When storing intermediates, MISTIQUE uses unique properties of traditional machine learning pipelines and deep neural networks to reduce the storage footprint of intermediates by 6X— 110X while reducing query execution time by up to 210X for DNNs and 390X for TRAD pipelines.

While this paper has made inroads into the problem of storing and querying model intermediates, we see multiple avenues for future work. First, this work focuses on efficient storage techniques as a way to speed up diagnostic queries. A parallel means of achieving this goal is to speed up query execution via techniques such as indexing, sampling, and approximation, and these approaches merit further investigation. Second, our query execution currently separates intermediate fetching from analysis; however, using knowledge about the analysis may allow us to make better decisions about retrieval (e.g., whether reconstruction is required or not). Similarly, although MISTIQUE currently optimizes access to intermediates on a per-query basis, a diagnosis session often involves many queries, and therefore there may be opportunities to further reduce execution time via caching and pre-fetching. Third, extending our work to other types of models, e.g., recurrent neural networks, may identify new opportunities for optimizations. Last, but not least, new techniques for model diagnosis that can leverage data stored in a system like MISTIQUE is a rich direction for research.

ACKNOWLEDGEMENTS

We thank the SIGMOD reviewers for their thoughtful feedback on this paper which led to significant improvements. Manasi Vartak is supported by the Facebook PhD Fellowship and Joana M. F. da Trindade is supported by an Alfred P. Sloan UCEM PhD Fellowship.

REFERENCES

[1] Anastassia Ailamaki, David J DeWitt, and Mark D Hill. 2002. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal - The International Journal on Very Large Data Bases* 11, 3 (2002), 198–215.

[2] Guillaume Alain and Yoshua Bengio. 2016. Understanding intermediate layers using linear classifier probes. *CoRR* abs/1610.01644 (2016). arXiv:1610.01644 http://arxiv.org/abs/1610.01644

[3] Saleema Amershi, Max Chickering, Steven M. Drucker, Bongshin Lee, Patrice Simard, and Jina Suh. 2015. ModelTracker: Redesigning Performance Analysis

Tools for Machine Learning. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 337–346.

[4] David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. 2017. Network Dissection: Quantifying Interpretability of Deep Visual Representations. In *Computer Vision and Pattern Recognition*.

[5] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. 2006. ULDBs: Databases with uncertainty and lineage. In *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 953–964.

[6] Anant Bhardwaj, Amol Deshpande, Aaron J. Elmore, David Karger, Sam Madden, Aditya Parameswaran, Harihar Subramanyam, Eugene Wu, and Rebecca Zhang. 2015. Collaborative Data Analytics with DataHub. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1916–1919. https://doi.org/10.14778/2824032.2824100

[7] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. 2015. Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1346–1357. https://doi.org/10.14778/2824032.2824035

[8] Souvik Bhattacherjee, Amol Deshpande, and Alan Sussman. 2014. Pstore: an efficient storage framework for managing scientific data. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*. ACM, 25.

[9] Rajendra Bose and James Frew. 2005. Lineage retrieval for scientific data processing: a survey. *ACM Computing Surveys (CSUR)* 37, 1 (2005), 1–28.

[10] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Proceedings of the 8th International Conference on Database Theory (ICDT '01)*. Springer-Verlag, London, UK, UK, 316–330.

[11] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos E Scheidegger, Cláudio T Silva, and Huy T Vo. 2006. VisTrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 745–747.

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages.

[13] Dong Chen, Rachel KE Bellamy, Peter K Malkin, and Thomas Erickson. 2016. Diagnostic visualization for non-expert machine learning practitioners: A design study. In *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*. IEEE, 87–95.

[14] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, USA, 785–794.

[15] Been Doshi-Velez, Finale; Kim. 2017. Towards A Rigorous Science of Interpretable Machine Learning. In *eprint arXiv:1702.08608*.

[16] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. 2017. Compressed linear algebra for large-scale machine learning. *The VLDB Journal* (12 Sep 2017). https://doi.org/10.1007/s00778-017-0478-1

[17] Reuben Feinman, Ryan R Curtin, Saurabh Shintre, and Andrew B Gardner. 2017. Detecting adversarial samples from artifacts. *arXiv preprint* (2017).

[18] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. *ICLR* (2015). http://arxiv.org/abs/1412.6572

[19] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015).

[20] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[21] Silu Huang, Liqi Xu, Jialin Liu, Aaron J Elmore, and Aditya Parameswaran. 2017. O rpheus DB: bolt-on versioning for relational databases. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1130–1141.

[22] Robert Ikeda and Jennifer Widom. 2010. Panda: A System for Provenance and Data. In *Proceedings of the 2Nd Conference on Theory and Practice of Provenance (TAPP'10)*. USENIX Association, Berkeley, CA, USA, 5–5.

[23] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: Data provenance support in spark. *Proceedings of the VLDB Endowment* 9, 3 (2015), 216–227.

[24] Minsuk Kahng, Pierre Andrews, Aditya Kalro, and Duen Horng Chau. 2017. ActiVis: Visual Exploration of Industry-Scale Deep Neural Network Models. *CoRR* abs/1704.01942 (2017).

[25] Pang Wei Koh and Percy Liang. 2017. Understanding Black-box Predictions via Influence Functions. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 1885–1894. http://proceedings.mlr.press/v70/koh17a.html

[26] Sanjay Krishnan and Eugene Wu. 2017. PALM: Machine Learning Explanations For Iterative Debugging. In *Proceedings of the 2Nd Workshop on Human-In-the-Loop Data Analytics (HILDA'17)*. ACM, New York, NY, USA, Article 4, 6 pages.

[27] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel. 2016. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Rec.* 44, 4 (May 2016), 17–22.

[28] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. 2006. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience* 18, 10 (2006), 1039–1065.

[29] Scott M Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 4768–4777.

[30] Michael Maddox, David Goehring, Aaron J Elmore, Samuel Madden, Aditya Parameswaran, and Amol Deshpande. 2016. Decibel: The relational dataset branching system. *Proceedings of the VLDB Endowment* 9, 9 (2016), 624–635.

[31] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. 2013. Ad Click Prediction: A View from the Trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '13)*. ACM, New York, NY, USA, 1222–1230.

[32] Dutch T. Meyer and William J. Bolosky. 2011. A Study of Practical Deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*. USENIX Association, Berkeley, CA, USA, 1–1.

[33] Hui Miao, Amit Chavan, and Amol Deshpande. 2017. ProvDB: Lifecycle Management of Collaborative Analysis Workflows.

[34] H. Miao, A. Li, L. S. Davis, and A. Deshpande. 2017. ModelHub: Deep Learning Lifecycle Management. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 1393–1394.

[35] Mahdi Milani Fard, Quentin Cormier, Kevin Canini, and Maya Gupta. 2016. Launch and Iterate: Reducing Prediction Churn. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 3179–3187.

[36] Peter Norvig. 2016. State-of-the-Art AI: Building Tomorrow's Intelligent Systems. (2016). http://events.technologyreview.com/video/watch/peter-norvig-state-of-the-art-ai/

[37] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. 2016. The TileDB array data storage manager. *Proceedings of the VLDB Endowment* 10, 4 (2016), 349–360.

[38] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 1–18.

[39] Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. 2017. SVCCA: Singular Vector Canonical Correlation Analysis for Deep Learning Dynamics and Interpretability. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 6078–6087.

[40] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, USA, 1135–1144.

[41] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[42] Adam Seering, Philippe Cudre-Mauroux, Samuel Madden, and Michael Stonebraker. 2012. Efficient Versioning for Scientific Array Databases. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering (ICDE '12)*. IEEE Computer Society, Washington, DC, USA, 1013–1024.

[43] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2016. Grad-cam: Visual explanations from deep networks via gradient-based localization. 7, 8 (2016).

[44] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning Important Features Through Propagating Activation Differences. *CoRR* abs/1704.02685 (2017).

[45] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2013. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *CoRR* abs/1312.6034 (2013). arXiv:1312.6034 http://arxiv.org/abs/1312.6034

[46] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[47] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. 2017. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 535–546. https://doi.org/10.1109/ICDE.2017.109

[48] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. VLDB Endowment, 553–564.

[49] Michael Stonebraker, Paul Brown, Jacek Becla, and Donghui Zhang. 2013. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engg.* 15, 3 (May 2013), 54–62. https://doi.org/10.1109/MCSE.2013.19
[50] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. 2016. ModelDB: A System for Machine Learning Model Management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA '16)*. ACM, New York, NY, USA, Article 14, 3 pages.
[51] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. 2013. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic acids research* 41, W1 (2013), W557–W561.
[52] Eugene Wu, Samuel Madden, and Michael Stonebraker. 2013. SubZero: A Fine-grained Lineage System for Scientific Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, Washington, DC, USA, 865–876.
[53] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. 2015. Understanding Neural Networks Through Deep Visualization. In *Deep Learning Workshop, International Conference on Machine Learning (ICML)*.
[54] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 15–28.
[55] Matthew D. Zeiler and Rob Fergus. 2014. *Visualizing and Understanding Convolutional Networks*. Springer International Publishing, Cham, 818–833. https://doi.org/10.1007/978-3-319-10590-1_53
[56] Zhao Zhang, Evan R. Sparks, and Michael J. Franklin. 2017. Diagnosing Machine Learning Pipelines with Fine-grained Lineage. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17)*. ACM, New York, NY, USA, 143–153.
[57] B. Zhou, A. Khosla, Lapedriza. A., A. Oliva, and A. Torralba. 2016. Learning Deep Features for Discriminative Localization. *CVPR* (2016).

A DETAILS OF DIAGNOSTIC TECHNIQUES

A.1 Visualizations

Figs. 12 and 13 respectively show screenshots of the visualization interfaces proposed in [24] and [13] respectively.

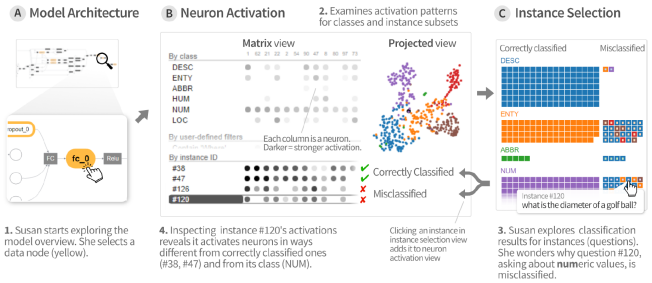


Figure 12: ActiVis Tool screenshot from [24]

A.2 SVCCA

Algorithm 1 presents pseudocode for the SVCCA technique proposed in [39].

A.3 NetDissect

Algorithm 2 presents pseudocode for the SVCCA technique proposed in [4].

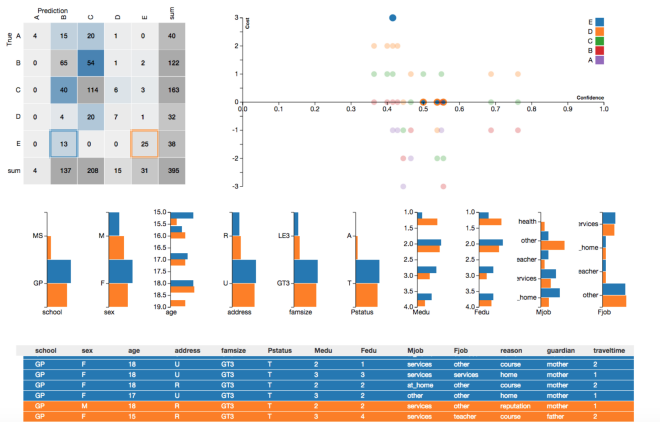


Figure 13: VizML Tool screenshot from [13]

Algorithm 1 SVCCA [39]

procedure SVCCA(A_{l_1}, A_{l_2}) // activations from layers l_1 and l_2
  $A'_{l_1} \leftarrow \text{SVD}(A_{l_1}, 0.99)$ // directions explaining 99% variance
  $A'_{l_2} \leftarrow \text{SVD}(A_{l_2}, 0.99)$ // as above
  $\{\rho_i, z^{l_1}, z^{l_2}\} \leftarrow \text{CCA}(A'_{l_1}, A'_{l_2})$ // set of canonical correlations
  return $\frac{\sum_i \rho_i}{min(size(l_1), size(l_2))}$

Algorithm 2 Netdissect [4]

procedure NETDISSECT(I, c, k, α) // images I, Concept c, unit k, activation threshold α
  $D_k \leftarrow A_k(I)$ // get activation maps for unit k
  $T_k \leftarrow percentile(D_k, 1-\alpha)$ // get threshold
  for Image x in I **do**
    $A(x) \leftarrow A_k(x)$
    for [i,j] in $A(x)$.cells **do** // binarize
      if $A(x)[i,j] \geq T_k$ **then**
        $A(x)[i,j] = 1$
      else
        $A(x)[i,j] = 0$
    $L(x) \leftarrow Labels_c(x)$ // pixel-wise label for concept
  return $\frac{\sum_x |L(x) \bigcap A(x)|}{\sum_x |L(x) \bigcup A(x)|}$ // intersection over union metric

B ALGORITHMS

Algorithm 3 and 4 respectively present the pseudocode for reading data from and writing data to MISTIQUE.

C ADDITIONAL EXPERIMENTS

C.1 Impact of quantization on KNN accuracy

C.2 Microbenchmark on Compression

Fig. 14 shows a microbenchmark illustrating the benefits of storing similar values together. We generated a 100K X 100 matrix of 32-bit floating point values with varying amounts of column similarity (0: completely different, 0.5: 50% of values are identical, 1: all values identical) and measured the storage footprint of storing the

Algorithm 3 Reading Data from MISTIQUE

```
1:  procedure GET_DATA(columns)
2:      dfs ← columns.parent_dataframe()
3:      tmp1 ← []
4:      for df ∈ dfs do
5:          tmp2 ← []
6:          for row_block ∈ df.row_blocks() do
7:              tmp3 ← []
8:              for column ∈ columns do
9:                  if column.is_materialized() then
10:                     partition ← column.read_partition()
11:                     tmp.append(partition[column])
12:                 else
13:                     rerun_pipeline(column, row_block)
14:             row_block_data ← h_concat(tmp3)
15:             tmp2.append(row_block_data)
16:         df_data = v_concat(tmp2)
17:         tmp1.append(df_data)
18:     res ← h_concat(tmp1)
19:     return res
```

Algorithm 4 Storing intermediates

```
1:  procedure STORE_INTERMEDIATE(row_block, γ_min)
2:      for col_chunk ∈ row_block.columns do
3:          stats ← get_stats(col_chunk)
4:          if γ < γ_min then // don't store
5:              update_stats(stats)
6:              return
7:          col_chunk ← quantize(col_chunk)
8:          identical_col ← get_identical_cols(col)
9:          if identical_col == nil then
10:             partition ← get_closest_partition(stats, sim)
11:             partition.add(col_chunk)
12:             evicted_partition ← bufferpool.add(partition)
13:             compress_and_store_partition(evicted_partition)
14:         else
15:             update(identical_col)
```

Layer	Full precision	8BIT_QT	POOL_QT ($\sigma = 2$)
11	1.0	0.94	0.74
16	1.0	0.96	0.84
19	1.0	1.0	1.0

Table 3: KNN accuracy: Fraction of overlap between true KNN and KNN computed across different storage schemes.

100 columns individually with gzip compression vs. storing them together with the same compression. From the figure, we can see that while the cost of individually compressed columns remains the same, storing similar columns together can produce a gain of up to 6X depending on column similarity.

D LAYER SIZES FOR CIFAR10_VGG16

E ZILLOW PIPELINE DETAILS

Here we describe the full pipelines used for the Zillow workflow. For this workload, we are given three csv files: (i) properties containing attributes of homes in the dataset; (ii) training data listing the property id, date of the property sale, and the error

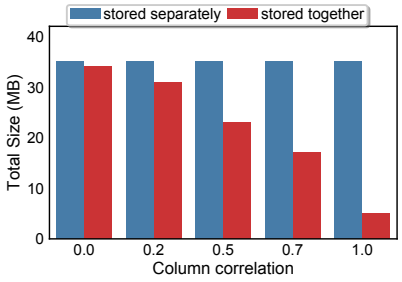


Figure 14: Column compression micro-benchmark.

Layer num	Name	Shape
0	input	(32, 32, 3)
1, 2	block1_conv1, block1_conv2	(32, 32, 64)
3	block1_pool	(16, 16, 64)
4, 5	block2_conv1, block2_conv2	(16, 16, 128)
6	block2_pool	(8, 8, 128)
7, 8, 9	block3_conv1	(8, 8, 256)
10	block3_pool	(4, 4, 256)
11, 12, 13	block4_conv1	(4, 4, 512)
14, 15, 16, 17	block4_pool	(2, 2, 512)
18	block5_pool	(1, 1, 512)
19	flatten	(512)
20	dense	(256)
21	logits	(10)

between the Zillow price estimate and the actual sale price; (iii) test data containing the property id and three dates of potential property sale. The goal is to predict the error at each of the potential dates. Table 4 shows all the pipeline template used in the Zillow workload. For each pipeline template, we generated 5 variations of the pipeline by choosing 5 different setting combinations for the listed hyperparameters. While we could have generated a much larger variations for each pipeline, we believe that the resulting set of 50 pipelines are sufficient to illustrate the advantages of MISTIQUE.

F EXPERIMENTAL QUERIES

In Table 5 we list the full queries used to evaluate MISTIQUE. For TRAD and DNN, we evaluate on two queries in each query category.

ID	Pipeline Template	Hyperparameters
P1	ReadCSV (3) → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainLightGBM → Predict (2)	learning_rate,sub_feature, min_data
P2	ReadCSV (3) → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainXGBoost → Predict (2)	eta,lambda,alpha,max_depth
P3	ReadCSV (3) → OneHotEncoding → FillNA (2) → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainEasticNet → Predict (2)	l1_ratio,tol
P4	ReadCSV (3) → Avg → OneHotEncoding → FillNA (2) → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainEasticNet → Predict (2)	l1_ratio,tol, normalize
P5	ReadCSV (3) → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainXGBoost,TrainElasticNet → Predict (4) → CombinePredictions (2)	eta,lambda,alpha,max_depth, xgb_weight, lgbm_weight
P6	ReadCSV (3) → Avg → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainLightGBM → Predict (2)	eta,lambda,alpha,max_depth, bagging_fraction
P7	ReadCSV (3) → Avg → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainXGBoost → Predict (2)	eta,lambda,alpha,max_depth, bagging_fraction
P8	ReadCSV (3) → Avg → GetConstructionRecency → OneHotEncoding → FillNA (2) → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainEasticNet → Predict (2)	l1_ratio,tol, normalize
P9	ReadCSV (3) → Avg → GetConstructionRecency → OneHotEncoding → FillNA (2) → ComputeNeighborhood → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainEasticNet → Predict (2)	ComputeNeighborhood_params, l1_ratio,tol, normalize
P10	ReadCSV (3) → Avg → GetConstructionRecency → OneHotEncoding → FillNA (2) → ComputeNeighborhood → IsResidential → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainEasticNet → Predict (2)	IsResidential_params, l1_ratio,tol, normalize

Table 4: Pipeline Templates for Zillow workload. The numbers in params indicate the number of times a transformation is applied (typically once on the training set and then again on test set)

Query Category	Specific instantiation	Intermediates Queried
Few Columns, Few Rows (FCFR)	(POINTQ) Find the activation map for neuron-35 in layer-4 for image-345.png	X, I
	(POINTQ) Find average lot size feature for for the Home-135 in P_{Oct31}	X, I
	(TOPK) Find the top-10 images that produce the highest activations for Neuron-35 in layer-13	X, I
	(TOPK) Find prediction error on the 10 homes that were most recently built	X, I
Few Columns, Many Rows (FCMR)	(COL_DIFF) Compare model performance for P_{Oct31} and P_{Nov1} grouped by type of house [31]	X, Y, P
	(COL_DIFF) Find the examples whose predictions differed between CIFAR10_CNN and CIFAR10_VGG16 [35]	X, Y_{cnn}, Y_{vgg}
	(COL_DIST) Plot the confidence score for all images predicted as cats [13]	X, I, Y, P
	(COL_DIST) Plot the error rates for all homes [13]	X, I, Y, P
Many Columns, Few Rows (MCFR)	(KNN) Find performance of CIFAR10_VGG16 for images similar to image-51	X, $x_{\text{img-51}}$, Y, P
	(KNN) Find predictions for the 10 homes most similar to Home-50	X, x_o, Y, P
	(ROW_DIFF) Compare the activations of neurons in layer-6 between an adversarial image and it's equivalent non-adversarial image	I, Y
	(ROW_DIFF) Compare features for Home-50 and Home-55 that are known to be in the same neighborhood but have very different prices	I, Y
Many Columns, Many Rows (MCMR)	(SVCCA) Find the features from interm-8 most correlated with the residual errors of P_{Oct31}	X, Y, P
	(SVCCA) Compute similarity between the logits of class ship and the representation learned by the last convolutional layer	I
	(VIS) Plot the average feature values for Victorian homes in Boston vs. Victorian homes in Seattle	
	(VIS) Plot the average activations for all neurons in layer-5 across all classes	I

Table 5: Experimental Queries