Deep Lake: a Lakehouse for Deep Learning

Sasun Hambardzumyan, Abhinav Tuli, Levon Ghukasyan, Fariz Rahman, Hrant Topchyan, David Isayan, Mark McQuade, Mikayel Harutyunyan, Tatevik Hakobyan, Ivo Stranic, Davit Buniatyan
team@activeloop.ai
Activeloop
Mountain View, CA, USA

ABSTRACT

Traditional data lakes provide critical data infrastructure for analytical workloads by enabling time travel, running SQL queries, ingesting data with ACID transactions, and visualizing petabyte-scale datasets on cloud storage. They allow organizations to break down data silos, unlock data-driven decision-making, improve operational efficiency, and reduce costs. However, as deep learning usage increases, traditional data lakes are not well-designed for applications such as natural language processing (NLP), audio processing, computer vision, and applications involving non-tabular datasets.

This paper presents Deep Lake, an open-source lakehouse for deep learning applications developed at Activeloop[1][2]. Deep Lake maintains the benefits of a vanilla data lake with one key difference: it stores complex data, such as images, videos, annotations, as well as tabular data, in the form of tensors and rapidly streams the data over the network to (a) Tensor Query Language, (b) in-browser visualization engine, or (c) deep learning frameworks without sacrificing GPU utilization. Datasets stored in Deep Lake can be accessed from PyTorch [58], TensorFlow [25], JAX [31], and integrate with numerous MLOps tools.

KEYWORDS

Deep Lake, Deep Learning, Data Lake, Lakehouse, Cloud Computing, Distributed Systems

1 INTRODUCTION

A data lake is a central repository that allows organizations to store structured, unstructured, and semi-structured data in one place. Data lakes provide a better way to manage, govern, and analyze data. In addition, they provide a way to break data silos and gain insights previously hidden in disparate data sources. First-generation data lakes traditionally collected data into distributed storage systems such as HDFS [71] or AWS S3 [1]. Unorganized collections of the data turned data lakes into "data swamps", which gave rise to the second-generation data lakes led by Delta, Iceberg, and Hudi [27, 15, 10]. They strictly operate on top of standardized structured formats such as Parquet, ORC, Avro [79, 6, 20] and provide features like time travel, ACID transactions, and schema evolution. Data lakes directly integrate with query engines such as Presto, Athena, Hive, and Photon [70, 12, 76, 66] to run analytical queries. Additionally, they connect to frameworks like Hadoop, Spark, and Airflow [14, 82, 9] for ETL pipeline maintenance. In its turn, the integration between data lakes and query engines with clear compute and storage separation resulted in the emergence of systems like Lakehouse [28] that serve as an alternative to data warehouses, including Snowflake, BigQuery, Redshift, and Clickhouse [33, 4, 40, 2].

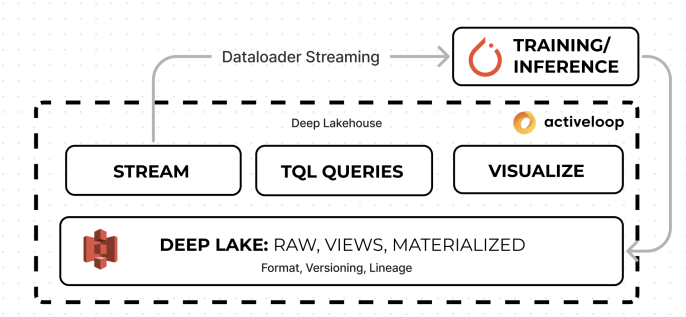

Figure 1: Deep Lake Architecture overview interfacing with deep learning frameworks.

Over the past decade, deep learning has outpaced traditional machine learning techniques involving unstructured and complex data such as text, images, videos, and audio [44, 47, 38, 83, 51, 30, 63, 56]. Not only did deep learning systems outgrow traditional techniques, but they also achieved super-human accuracy in applications such as cancer detection from X-ray images, anatomical reconstruction of human neural cells, playing games, driving cars, unfolding proteins, and generating images [61, 48, 72, 42, 77]. Large language models with transformer-based architectures achieved state-of-the-art results across translation, reasoning, summarization, and text completion tasks [78, 36, 81, 32]. Large multi-modal networks embed unstructured data into vectors for cross-modal search [29, 60]. Moreover, they are used to generate photo-realistic images from text [62, 65].

Although one of the primary contributors to the success of deep learning models has been the availability of large datasets such as CoCo (330K images), ImageNet (1.2M images), Oscar (multilingual text corpus), and LAION (400M and 5B images) [49, 34, 74, 68], it does not have a well-established data infrastructure blueprint similar to traditional analytical workloads to support such scale. On the other hand, Modern Data Stack (MDS) lacks the features required to deploy performant deep learning-based solutions so organizations opt to develop in-house systems.

In this paper, we introduce Deep Lake, a lakehouse specialized for deep learning workloads. Deep Lake retains the main benefits of a

[1]Source code available: https://github.com/activeloopai/deeplake
[2]Documentation available at https://docs.deeplake.ai

 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

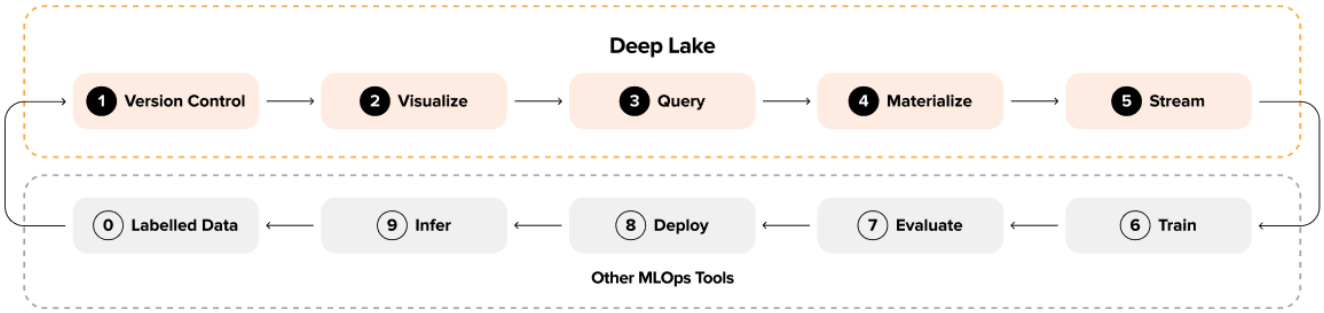

Figure 2: Machine Learning Loop with Deep Lake

traditional data lake with one notable distinction: it stores complex data, such as images, videos, annotations, and tabular data, as tensors and rapidly streams the data to deep learning frameworks over the network without sacrificing GPU utilization. Furthermore, it provides native interoperability between deep learning frameworks such as PyTorch, TensorFlow, and JAX [58, 25, 31].

The main technical contributions of this paper include:

Tensor Storage Format that stores dynamically shaped arrays on object storage;
Streaming Dataloader that schedules fetching, decompression, and user-defined transformations, optimizing data transfer throughput to GPUs for deep learning;
Tensor Query Language running SQL-like operations on top of multi-dimensional array data;
In-browser visualization engine that streams data from object storage and renders it in the browser using WebGL.

The remainder of this paper unfolds as follows. We begin by considering current challenges in deep learning on unstructured data. Next, we present the Tensor Storage Format (TSF) with its key concepts. Furthermore, we discuss Deep Lake's capabilities and applications within the ML cycle. Next, we provide performance experiments and discuss the results. Finally, we review related work, list possible limitations, and conclude.

2 CURRENT CHALLENGES

In this section, we discuss the current and historical challenges of unstructured or complex data management.

2.1 Complex Data Types in a Databases

It is generally not recommended to store binary data, such as images, directly in a database. This is because databases are not optimized for storing and serving large files and can cause performance issues. In addition, binary data does not fit well into a database's structured format, making it difficult to query and manipulate. This can lead to slow load times for users. Databases are typically more expensive to operate and maintain than other types of storage, such as file systems or cloud storage services. Therefore, storing large amounts of binary data in a database can be more costly than other storage solutions.

2.2 Complex Data Along with Tabular Formats

Increases in large-scale analytical and BI workloads motivated the development of compressed structured formats like Parquet, ORC, Avro, or transient in-memory formats like Arrow [79, 6, 20, 13]. As tabular formats gained adoption, attempts to extend those formats, such as Petastorm [18] or Feather [7] for deep learning, have emerged. To the best of our knowledge, these formats have yet to gain wide adoption. This approach primarily benefits from native integrations with Modern Data Stack (MDS). However, as discussed previously, upstream tools require fundamental modifications to adapt to deep learning applications.

2.3 Object Storage for Deep Learning

The current cloud-native choice for storing large unstructured datasets is object storage such as AWS S3 [1], Google Cloud Storage (GCS) [3], or MinIO [17]. Object storage does offer three main benefits over distributed network file systems. They are (a) cost-efficient, (b) scalable, and (c) serve as a format-agnostic repository. However, cloud storages are not without drawbacks. Firstly, they introduce significant latency overhead, especially when iterating over many small files such as text or JSON. Next, unstructured data ingestion without metadata control can produce "data swamps". Furthermore, object storage has built-in version control; it is rarely used in data science workflows. Lastly, data on object storage gets copied to a virtual machine before training, thus resulting in storage overhead and additional costs.

2.4 Second Generation of Data Lakes

The second-generation data lakes led by Delta, Iceberg, Hudi [27, 15, 10] extend object storage by managing tabular format files with the following primary properties.

(1) *Update operations*: inserting or deleting a row on top of a tabular format file.
(2) *Streaming*: downstream data ingestion with ACID properties and upstream integration with query engine exposing SQL interface.
(3) *Schema evolution*: evolving columnar structure while preserving backward compatibility.
(4) *Time travel and audit log trailing*: preserving historical state with rollback property where queries can be reproducible. Also, support for row-level control on data lineage.

(5) *Layout optimization*: Built-in feature to optimize file sizes and data compaction with custom ordering support. Significantly speeds up querying.

However, second-generation data lakes are still bound by the limitations of the inherent data formats to be used in deep learning, as previously discussed in section 2.2. Hence in this paper, we extend the second generation of data lake capabilities for deep learning use cases by rethinking the format and upstream features, including querying, visualization, and native integration to deep learning frameworks to complete the ML lifecycle as shown in Fig. 2.

3 TENSOR STORAGE FORMAT

Deep Lake datasets follow columnar storage architecture, with tensors as columns, as shown in Fig. 3. Each tensor is a collection of *chunks* - binary blobs that contain the data samples. An index map associated with each tensor helps find the right chunk and index of the sample within that chunk for a given sample index.

3.1 Dataset

A sample in a dataset represents a single row indexed across parallel tensors. As opposed to a document storage format, sample elements are logically independent, which enables partial access to samples for running performant queries or streaming selected tensors over the network to the GPU training instances. Multiple tensors can be grouped. Groups implement syntactic nesting and define how tensors are related to each other. Syntactic nesting avoids the format complication for hierarchical memory layout. Changes to the dataset's schema are also tracked over time with version control, similar to dataset content changes.

3.2 Tensors

Tensors are typed and can be appended or modified in-place. Default access to an index or a set of indices returns the data as NumPy arrays [55]. Instead of storing 1-D data as seen in Parquet [79] or series in Arrow [13], tensors can accommodate n-dimensional data, where typically the first dimension corresponds to the index or batch dimension. Tensors can contain dynamically shaped arrays, also called ragged tensors, as opposed to other statically chunked array formats such as Zarr [52].

3.3 Types

Htype defines the expectations on samples in a tensor such as data type (*dtype* as seen in NumPy [55]), shape, number of dimensions, or compression. Typed tensors make interacting with deep learning frameworks straightforward and enable sanity checks and efficient memory layout. By inheriting from a generic tensor htype, we can construct types such as *image*, *video*, *audio*, *bbox*, *dicom*, and others. For example, a tensor with *image* htype would expect samples being appended to it to have *dtype* as *uint8* and shape length 3 (i.e. width, height and number of channels). We further expand on the notion of htypes allowing for meta types that support storing image sequences in tensors (*sequence[image]*), referencing to remotely stored images, while maintaining the regular behavior of a *image* tensor (*link[image]*), or even possible cross-format support.

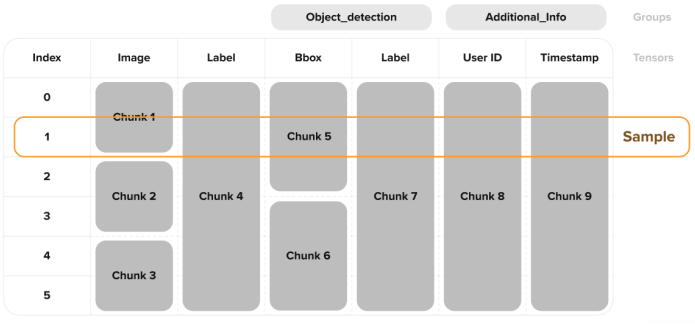

Figure 3: How each sample (row) is stored in a set of columnar tensors with dynamically sized chunks

3.4 Memory Layout

A Deep Lake dataset contains a provenance file in JSON format and folders per tensor. A tensor contains chunks, chunk encoder, tile encoder, and tensor metadata. Tensors can be optionally hidden. For instance, hidden tensors can be used to maintain down-sampled versions of images or preserve shape information for fast queries.

Tensors are stored in chunks at the storage level. While statically (inferred) shaped chunking avoids maintaining a chunk map table, it introduces significant user overhead during the specification of the tensor, custom compression usage limitations, underutilized storage for dynamically shaped tensors, and post-processing inefficiencies. Deep Lake chunks are constructed based on the lower and upper bound of the chunk size to fit a limited number of samples. This comes with a trade-off of having a compressed index map that preserves the sample index to chunk id mapping per tensor while enabling chunk sizes in the range optimal for streaming while accommodating mixed-shape samples. One could consider the approach taken in this paper as an optimized trade-off between file system page map and compute-defined map-less array storage system. For practical reasons, a single chunk encoder can be scaled to billions of images while maintaining a 150MB chunk encoder per 1PB tensor data. Further scaling can be introduced by sharding the chunk encoder. Chunks contain header information such as byte ranges, shapes of the samples, and the sample data itself. If a sample is larger than the upper bound chunk size, which is the case for large aerial or microscopy images, the sample is tiled into chunks across spatial dimensions. The only exception to tiling is videos. Videos are preserved due to efficient frame mapping to indices, key-frame-only decompression, and range-based requests while streaming.

3.5 Access Patterns

The tensor storage format is optimized for deep learning training and inference, including sequential and random access. Sequential access is used for running scan queries, transforming tensors into other tensors, or running inference. Random access use cases include multiple annotators writing labels to the same image or models storing back predictions along with the dataset. While the strict mode is disabled, out-of-the-bounds indices of a tensor can be assigned, thus accommodating sparse tensors. However, random assignment over time will produce inefficiently stored data chunks.

To fix the data layout, we implement an on-the-fly re-chunking algorithm to optimize the data layout. One of the key access patterns of Deep Lake is shuffled stream access for training machine learning models. It requires random or custom order access while streaming chunks into the training process. This is achieved by involving range-based requests to access sub-elements inside chunks, running complex queries before training to determine the order, and maintaining a buffer cache of fetched and unutilized data. This avoids having a separate compute cluster for running shuffling algorithm [50].

Each tensor has its own chunks, and the default chunk size is 8MB. A single chunk consists of data from multiple indices when the individual data points (image, label, annotation, etc.) are smaller than the chunk size. Conversely, when individual data points are larger than the chunk size, the data is split among multiple chunks (tiling). Exceptions to chunking logic are video data.

Deep Lake format is optimized for maximizing throughput to GPU processing. It includes CPU pre-fetching, decompression or decoding, transformations, and GPU memory transfer in a deep learning framework's expected layout.

3.6 Storage Providers

Deep Lake can be plugged into any storage provider, including object storages such as AWS S3 [1], Google Cloud Storage (GCS) [3], POSIX compatible file systems, or local in-memory storage. Moreover, it constructs memory caching by chaining various storage providers together, for instance - the Least Recently Used (LRU) cache of remote S3 storage with local in-memory data.

4 DEEP LAKE SYSTEM OVERVIEW

As shown in Fig. 1, Deep Lake stores raw data and views in object storage such as S3 and materializes datasets with full lineage. Streaming, Tensor Query Language queries, and Visualization engine execute along with either deep learning compute or on the browser without requiring external managed or centralized service.

4.1 Ingestion

4.1.1 Extract. Sometimes metadata might already reside in a relational database. We additionally built an ETL destination connector using Airbyte[3] [22]. The framework allows plugging into any supported data source, including SQL/NoSQL databases, data lakes, or data warehouses, and synchronizing the data into Deep Lake. Connector protocol transforms the data into a columnar format.

4.1.2 Transform. To significantly accelerate data processing workflows and free users from worrying about the chunk layout, Deep Lake provides an option to execute python transformations in parallel. The transformation takes in a dataset, sample-wise iterates across the first dimension, and outputs a new dataset. A user defined python function expects two required arguments *sample_in*, *sample_out* and is decorated with *@deeplake.compute*. A single *sample_in* can dynamically create multiple *sample_outs*. It enables both one-to-one and one-to-many transformations. The transformation can also be applied in place without creating a new dataset.

[3]Source code available: https://github.com/activeloopai/airbyte on the branch @feature/connector/deeplake

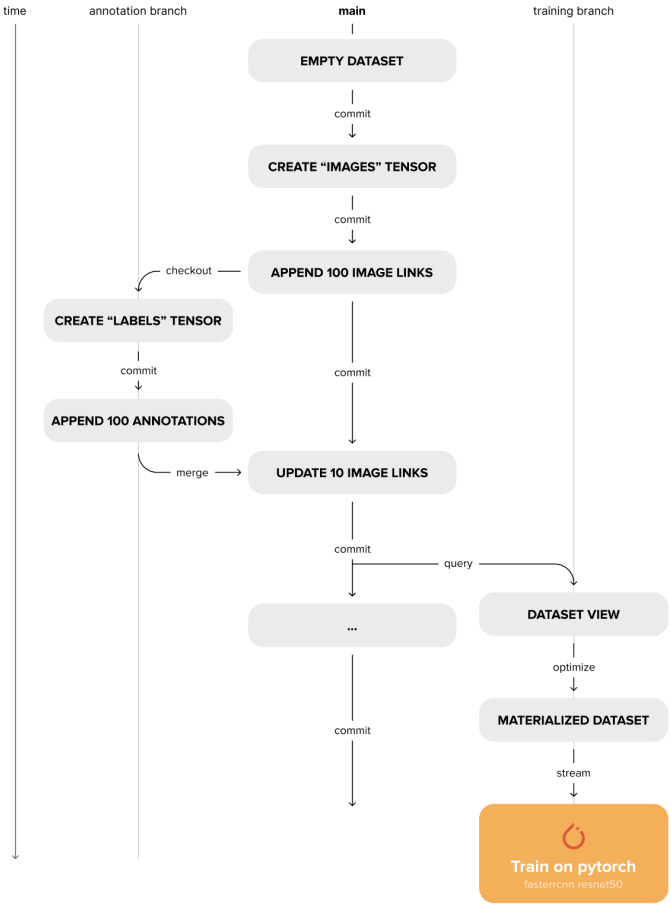

Figure 4: Version History of Evolving Deep Lake Dataset from empty till materialized view

Behind the scenes, the scheduler batches sample-wise transformations operating on nearby chunks and schedule them on a process pool. Optionally, the compute can be delegated to a Ray cluster [53]. Instead of defining an input dataset, the user can provide an arbitrary iterator with custom objects to create ingestion workflows. Users can also stack together multiple transformations and define complex pipelines.

4.2 Version Control

Deep Lake also addresses the need for the reproducibility of experiments and compliance with a complete data lineage. Different versions of the dataset exist in the same storage, separated by sub-directories. Each sub-directory acts as an independent dataset with its metadata files. Unlike a non-versioned dataset, these sub-directories only contain chunks modified in the particular version, along with a corresponding chunk_set per tensor containing the names of all the modified chunks. A version control info file present at the root of the directory keeps track of the relationship between these versions as a branching version-control tree. While accessing any chunk of a tensor at a particular version, the version control tree is traversed starting from the current commit, heading towards

the first commit. During the traversal, the chunk set of each version is checked for the existence of the required chunk. If the chunk is found, the traversal is stopped, and data is retrieved. For keeping track of differences across versions, for each version, a commit diff file is also stored per tensor. This makes it faster to compare across versions and branches. Moreover, the ids of samples are generated and stored during the dataset population. This is important for keeping track of the same samples during merge operations. Deep Lake's version control interface is the Python API, which enables machine learning engineers to version their datasets within their data processing scripts without switching back and forth from the CLI. It supports the following commands:

Commit: creates an immutable snapshot of the current state of the dataset.
Checkout: checks out to an existing branch/commit or creates a new branch if one doesn't exist.
Di : compares the differences between 2 versions of the dataset.
Merge: merges two different versions of the dataset, resolving conflicts according to the policy defined by the user.

4.3 Visualization of Tensors

Data visualization is a crucial part of ML workflows, especially when the data is hard to parse analytically. Fast and seamless visualization allows faster data collection, annotation, quality inspection, and training iterations. The Deep Lake visualizer engine provides a web interface for visualizing large-scale data directly from the source. It considers *htype* of the tensors to determine the best layout for visualization. Primary tensors, such as *image*, *video* and *audio* are displayed first, while secondary data and annotations, such as *text*, *class_label*, *bbox* and *binary_mask* are overlayed. The visualizer also considers the meta type information, such as *sequence* to provide a sequential view of the data, where sequences can be played and jump to the specific position of the sequence without fetching the whole data, which is relevant for video or audio use cases. Visualizer addresses critical needs in ML workflows, enabling users to understand and troubleshoot the data, depict its evolution, compare predictions to ground truth or display multiple sequences of images (e.g., camera images and disparity maps) side-by-side.

4.4 Tensor Query Language

Querying and balancing datasets is a common step in training deep learning workflows. Typically, this is achieved inside a dataloader using sampling strategies or separate pre-processing steps to subselect the dataset. On the other hand, traditional data lakes connect to external analytical query engines [66] and stream Dataframes to data science workflows. To resolve the gap between the format and fast access to the specific data, we provide an embedded SQL-like query engine implemented in C++ called Tensor Query Language (TQL). An example query is shown at Fig. 5. While SQL parser has been extended from Hyrise [37] to design Tensor Query Language, we implemented our planner and execution engine that can optionally delegate computation to external tensor computation frameworks. The query plan generates a computational graph of tensor operations. Then the scheduler, executes the query graph.

```
SELECT
  images[100:500, 100:500, 0:2] as crop,
  NORMALIZE(
    boxes,
    [100, 100, 400, 400]) as box
FROM
  dataset
WHERE IOU(boxes, "training/boxes") > 0.95
ORDER BY IOU(boxes, "training/boxes")
ARRANGE BY labels
```

Figure 5: An example query that arranges cropped images ordered by bounding boxes predictions error measured over user-defined function IOU (Intersection over Union).

Execution of the query can be delegated to external tensor computation frameworks such as PyTorch [58] or XLA [64] and efficiently utilize underlying accelerated hardware. In addition to standard SQL features, TQL also implements numeric computation. There are two main reasons for implementing a new query language. First, traditional SQL does not support multidimensional array operations such as computing the mean of the image pixels or projecting arrays on a specific dimension. TQL solves this by adding Python/NumPy-style indexing, slicing of arrays, and providing a large set of convenience functions to work with arrays, many of which are common operations supported in NumPy. Second, TQL enables deeper integration of the query with other features of the Deep Lake, such as version control, streaming engine, and visualization. For example, TQL allows querying data on specific versions or potentially across multiple versions of a dataset. TQL also supports specific instructions to customize the visualization of the query result or seamless integration with the dataloader for filtered streaming. The embedded query engine runs along with the client either while training a model on a remote compute instance or in-browser compiled over WebAssembly. TQL extends SQL with numeric computations on top of multi-dimensional columns. It constructs views of datasets, which can be visualized or directly streamed to deep learning frameworks. Query views, however, can be sparse, which can affect streaming performance.

4.5 Materialization

Most of the raw data used for deep learning is stored as raw files (compressed in formats like JPEG), either locally or on the cloud. A common way to construct datasets is to preserve pointers to these raw files in a database, query this to get the required subset of data, fetch the filtered files to a machine, and then train a model iterating over files. In addition, data lineage needs to be manually maintained with a provenance file. Tensor Storage Format simplifies these steps using linked tensors - storing pointers (links/urls to one or multiple cloud providers) to the original data. The pointers within a single tensor can be connected to multiple storage providers, thus allowing users to get a consolidated view of their data present in multiple sources. All of Deep Lake's features including queries, version control, and streaming to deep learning frameworks can be used with linked tensors. However, the performance of data streaming will not be as optimal as default tensors. A similar problem exists with

sparse views created due to queries, which would be inefficiently streamed due to the chunk layout. Furthermore, materialization transforms the dataset view into an optimal layout to stream into deep learning frameworks to iterate faster. Materialization involves fetching the actual data from links or views and efficiently laying it out into chunks. Performing this step towards the end of machine learning workflows leads to minimum data duplication while ensuring optimal streaming performance and minimal data duplication, with full data lineage.

4.6 Streaming Dataloader

As datasets become larger, storing and transferring over the network from a remotely distributed storage becomes inevitable. Data streaming enables training models without waiting for all of the data to be copied to a local machine. The streaming dataloader ensures data fetching, decompression, applying transformations, collation, and data handover to the training model. Deep learning dataloaders typically delegate fetching and transformation to parallel running processes to avoid synchronous computation. Then the data is transferred to the main worker through inter-process communication (IPC) which introduces memory copy overhead or uses shared memory with some reliability issues. In contrast, Deep Lake dataloader delegates highly parallel fetching and in-place decompressing in C++ per process to avoid global interpreter lock. Then, it passes the in-memory pointer to the user-defined transformation function and collates before exposing them to the training loop in deep learning native memory layout. Transformation concurrently executes in parallel when it uses only native library routine calls and releases python global interpreter lock (GIL) accordingly. As a result, we get:

Performance: Delivering data to the deep learning model fast enough so that either the GPU is fully utilized or bottlenecked by the compute.
Smart Scheduler: Dynamically differentiating between CPU-intensive jobs prioritization over less-intensive.
E cient Resource Allocation: Predicting memory consumption to avoid breaking the training process due to memory overfilling.

5 MACHINE LEARNING USE CASES

In this section, we review the applications of Deep Lake.

A typical scenario in a Deep Learning application starts with

(1) A raw set of files that is collected on an object storage bucket. It might include images, videos, and other types of multimedia data in their native formats such as JPEG, PNG or MP4.
(2) Any associated metadata and labels stored on a relational database. Optionally, they could be stored on the same bucket along with the raw data in a normalized tabular form such as CSV, JSON, or Parquet format.

As shown in Fig. 4, an empty Deep Lake dataset is created. Then, empty tensors are defined for storing both raw data as well as metadata. The number of tensors could be arbitrary. A basic example of an image classification task would have two tensors,

images tensor with htype of *image* and sample compression of JPEG
labels tensor with htype of *class_label* and chunk compression of LZ4.

After declaring tensors, the data can be appended to the dataset. If a raw image compression matches the tensor sample compression, the binary is directly copied into a chunk without additional decoding. Label data is extracted from a SQL query or CSV table into a categorical integer and appended into *labels* tensor. *labels* tensor chunks are stored using LZ4 compression. All Deep Lake data is stored in the bucket and is self-contained. After storage, the data can be accessed in a NumPy interface or as a streamable deep learning dataloader. Then, the model running on a compute machine iterates over the stream of image tensors, and stores the output of the model in a new tensor called *predictions*. Furthermore, we discuss below how one can train, version control, query, and inspect the quality of a Deep Lake dataset.

5.1 Deep Learning Model Training

Deep learning models are trained at multiple levels in an organization, ranging from exploratory training occurring on personal computers to large-scale training that occurs on distributed machines involving many GPUs. The time and effort required to bring the data from long-term storage to the training client are often comparable to the training itself. Deep Lake solves this problem by enabling rapid streaming of data without bottlenecking the downstream training process, thus avoiding the cost and time required to duplicate data on local storage.

5.2 Data Lineage and Version Control

Deep learning data constantly evolve as new data is added and existing data is quality controlled. Analytical and training workloads occur in parallel while the data is changing. Hence, knowing which data version was used by a given workload is critical to understand the relationship between the data and model performance. Deep Lake enables deep learning practitioners to understand which version of their data was used in any analytical workload and to time travel across these versions if an audit is required. Since all data is mutable, it can be edited to meet compliance-related privacy requirements. Like Git for code, Deep Lake also introduces the concept of data branches, allowing experimentation and editing of data without affecting colleagues' work.

5.3 Data Querying and Analytics

Training of deep learning models rarely occurs on all data collected by an organization for a particular application. Training datasets are often constructed by filtering the raw data based on conditions increasing model performance, which often includes data balancing, eliminating redundant data, or selecting data that contains specific features. Deep Lake provides the tools to query and analyze data so that deep learning engineers can create datasets yielding the highest accuracy models.

5.4 Data Inspection and Quality Control

Though unsupervised learning is becoming more applicable in real-world use cases, most deep learning applications still rely on supervised learning. Any supervised learning system is only as good as the quality of its data, often achieved by manual and exhaustive inspection of the data. Since this process is time-consuming, it is critical to provide the humans in the loop with tools to examine vast amounts of data very quickly. Deep Lake allows inspecting deep learning datasets of any size from the browser without any setup time or need to download data. Furthermore, the tools can be extended for comparing model results with ground truth. Combined with querying and version control, this can be applied to the iterative improvement of data to achieve the best possible model.

6 PERFORMANCE BENCHMARKS

In this section, we experimentally demonstrate Deep Lake's performance at scale from the point of ingestion into the format up to training at scale against other dataloaders and formats. We compare streaming datasets from different storage backends, and showcase performance gains and scalability while training on the cloud.

6.1 Ingestion speed to various formats

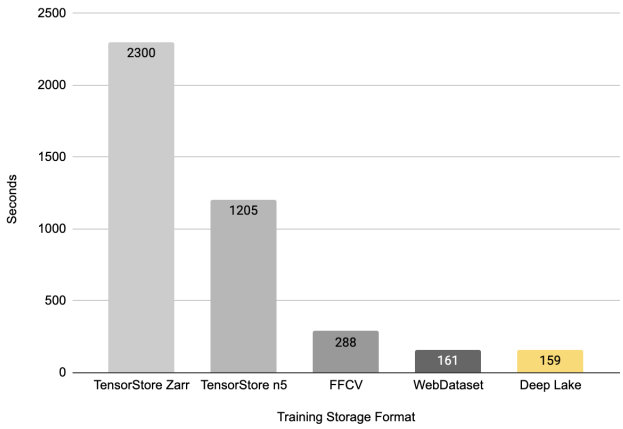

Figure 6: Ingesting 10,000 images from FFHQ [43] dataset into different format (lower better)

10,000 images from FFHQ [43] dataset were uncompressed and stored in NumPy format. Each 1024x1024x3 raw image is a 3MB array. Then, as shown in Fig. 6 images were serially written into each format. To increase the performance, we used TensorStore [23] to write to Zarr [52] and N5 [24] formats. The experiments were done on the AWS c5.9xlarge machine. Deep Lake achieves significantly faster write performance compared to array formats and on par with binary formats such as WebDataset [19] and FFCV Beton [39]

6.2 Comparison of local dataloaders

As shown in Fig. 7 Deep Lake achieves faster data loading in a PyTorch training loop without a model. The experiment was carried out on AWS P3.2xlarge instance with one Nvidia V100 GPU

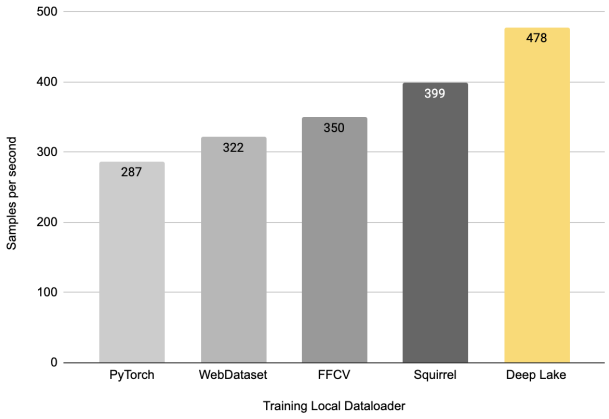

Figure 7: Iteration speed of images against other dataloaders (higher better)

card. The dataset has randomly generated 50,000 250x250x3 images stored as JPEG files. The list of libraries in which the benchmarks were carried out was Deep Lake, FFCV [39], Squirrel [75], Webdataset [19] and native PyTorch dataloader [58].

6.3 Streamable dataloader from different locations

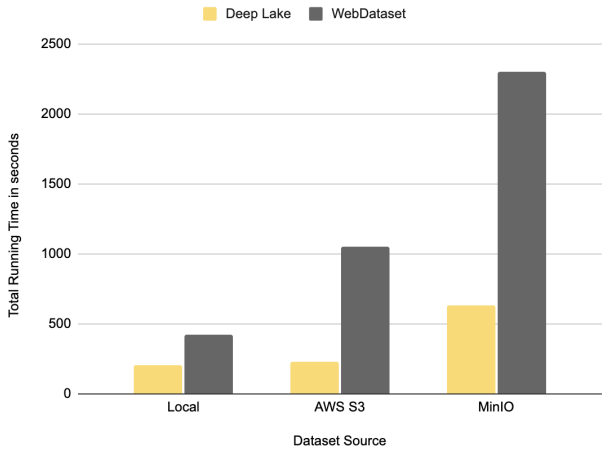

Figure 8: Streaming from different data storage locations: Local FileSystem, AWS S3, MinIO (lower better)

In this experiment as shown in Fig. 8, we explore different storage backends for remote streaming using the same dataset as in Section 6.2. MinIO [17] is running on another machine in a local network. Notably, Deep Lake achieves similar performance as if the data is local to the machine compared to AWS S3. Both WebDataset and Deep Lake are significantly slower while streaming the data from

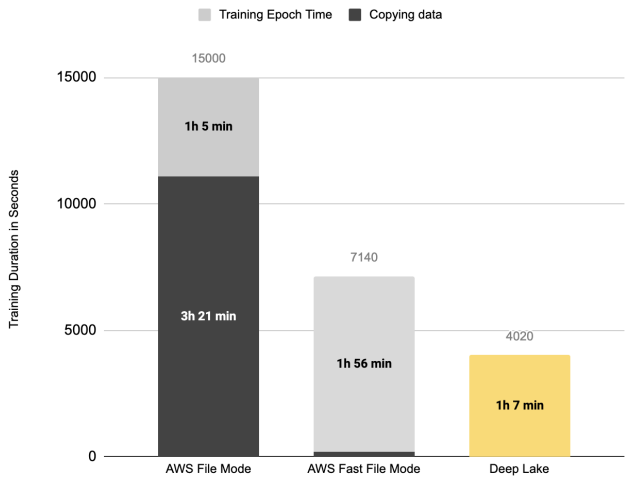

Figure 9: Training on ImageNet on an S3: AWS File Mode copies file by file from S3; Fast File Mode starts immediately with slower training; Deep Lake performs as if data is local, although it is streamed (lower better)

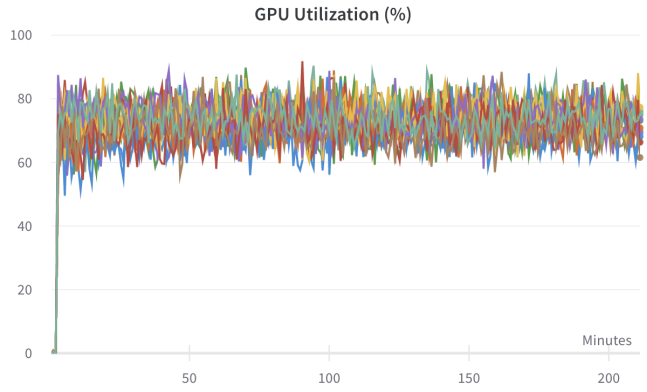

Figure 10: GPU utilization of single 16xA100 GPU machine while training 1B parameter CLIP model [60]. The dataset is LAION-400M [68] streaming from AWS us-east to GCP us-central datacenter. Each color demonstrates single A100 GPU utilization over training.

MinIO compared to AWS S3. For more detailed dataloader benchmarks, we would recommend an exhaustive dataloader overview study by Ofeidis et al. [54].

6.4 ImageNet training on the cloud

Since Deep Lake was built to be cloud-first, in this and next section we demonstrate the benefits it provides for training models on the cloud. We take ImageNet dataset [35] and store it on AWS S3 [1] as original and Tensor Storage Format. The dataset contains 1.2 million images and labels in total 150GB. Deep Lake achieves virtually similar training performance as if the data were local to the machine. This saves up to 4x GPU compute time and cost as shown in Fig. 9

6.5 Distributed training of a large multi-modal dataset

As a second experiment, we take LAION dataset [67] containing 400M image-text pairs and train CLIP [60], image-text embedding model with 1 billion parameters. The original dataset is a table of Parquet files with a column of image URLs. The dataset download from the source took 100 hours, while ingestion to Tensor Storage format took only 6 hours, totaling 1.9TB in size. The dataset has been stored on AWS in the US-east region while training GPU machine in the US-central region. As shown on Fig. 10 Deep Lake achieves high GPU utilization by streaming 5,100 images/s into 16 Nvidia A100 GPUs while without model up to 80,000 images/s per machine on the same region.

7 DISCUSSION AND LIMITATIONS

Deep Lake's primary use cases include (a) Deep Learning Model Training, (b) Data Lineage and Version Control, (c) Data Querying, and Analytics, (d) Data Inspection and Quality Control. We took NumPy [55] arrays as a fundamental block and implemented version control, streaming dataloaders, visualization engine from scratch.

7.1 Format Design Space

The Tensor Storage Format (TSF) is a binary file format designed specifically for storing tensors, which are multi-dimensional arrays of numerical values used in many machine learning and deep learning algorithms. The TSF format is designed to be efficient and compact, allowing for fast and efficient storage and access of tensor data. One key advantage of the TSF format is that it supports a wide range of tensor data types, including dynamically shaped tensors.

In comparison, the Parquet [79] and Arrow [13] formats are columnar file formats that are designed for storing and processing large analytical datasets. Unlike TSF, which is specifically designed for tensor data, Parquet and Arrow are optimized for efficient storage and querying of analytical workloads on tabular and time-series data. They use columnar storage and compression techniques to minimize storage space and improve performance, making them suitable for big data applications. However, TSF has some advantages over Parquet and Arrow when it comes to tensor data. TSF can support tensor operations and efficient streaming to deep learning frameworks.

Other tensor formats [18, 52, 23, 57] are efficient for massively massively parallelizable workloads as they don't require coordination across chunks. Tensor Storage Format key trade-off is enabling to store dynamically shape arrays inside a tensor without padding memory footprint. For example, in computer vision it is very common to store multiple images with different shapes or videos have dynamic length. To support the flexibility, minor overhead is introduced in the form of previously discussed chunk encoder that in practice we haven't observed impact on production workloads.

7.2 Dataloader

Deep Lake achieves state-of-the-art results in local and remote settings, as seen in benchmarks for iterating on large images Fig. 7. Primarily, it has been faster than FFCV [39], which claimed a reduction of ImageNet model training up to 98 cents per model training. Furthermore, Deep Lake achieves similar ingestion performance to WebDataset [19]. Deep Lake significantly outperforms on larger images. Parquet is optimized for small cells and analytical workloads, while Deep Lake is optimized for large, dynamically shaped tensorial data. Compared to other data lake solutions, its minimal python package design enables Deep Lake to be easily integrated into large-scale distributed training or inference workloads.

7.3 Future work

The current implementation of Deep Lake has opportunities for further improvement. Firstly, the storage format does not support custom ordering for an even more efficient storage layout required for vector search or key-value indexing. Secondly, Deep Lake implements branch-based locks for concurrent access. Similar to Delta ACID transaction model [27], Deep Lake can be extended to highly-performant parallel workloads. Thirdly, the current implementation of TQL only supports a subset of SQL operations (i.e., does not support operations such as *join*). Further work will focus on making it SQL-complete, extending to more numeric operations, running federated queries in external data sources and benchmarking against SQL engines.

8 RELATED WORK

Multiple projects have tried to improve upon or create new formats for storing unstructured datasets including TFRecord extending Protobuf [5], Petastorm [18] extending Parquet [79], Feather [7] extending arrow [13], Squirrel using MessagePack [75], Beton in FFCV [39]. Designing a universal dataset format that solves all use cases is very challenging. Our approach was mostly inspired by CloudVolume [11], a 4-D chunked NumPy storage for storing large volumetric biomedical data. There are other similar chunked NumPy array storage formats such as Zarr [52], TensorStore [23], TileDB [57]. Deep Lake introduced a typing system, dynamically shaped tensors, integration with fast deep learning streaming data loaders, queries on tensors and in-browser visualization support. An alternative approach to store large-scale datasets is to use HPC distributed file system such as Lustre [69], extending with PyTorch cache [45] or performant storage layer such as AIStore [26]. Deep Lake datasets can be stored on top of POSIX or REST API-compatible distributed storage systems by leveraging their benefits. Other comparable approaches evolve in vector databases [80, 8, 80] for storing embeddings, feature stores [73, 16] or data version control systems such as DVC [46], or LakeFS [21]. In contrast, Deep Lake version control is in-built into the format without an external dependency, including Git. Tensor Query Language, similar to TQP [41] and Velox [59] approaches, runs n-dimensional numeric operations on tensor storage by truly leveraging the full capabilities of deep learning frameworks. Overall, Deep Lake takes parallels from data lakes such as Hudi, Iceberg, Delta [27, 15, 10] and complements systems such as Databarick's Lakehouse [28] for Deep Learning applications.

9 CONCLUSION

We presented Deep Lake, the lakehouse for deep learning. Deep Lake is designed to help deep learning workflows run as seamlessly as analytical workflows run on Modern Data Stack. Notably, Deep Lake is built to retain prominent features of data lakes, such as time travel, querying, and rapid data ingestion at scale. One important distinction from traditional data lakes is Deep Lake's ability to store unstructured data with all its metadata in deep learning-native columnar format, which enables rapid data streaming. This allows materializing data subsets on-the-fly, visualizing them in-browser, or ingesting them into deep learning frameworks without sacrificing GPU utilization. Finally, we show that Deep Lake achieves state-of-the-art performance for deep learning on large datasets via multiple benchmarks.

10 ACKNOWLEDGEMENT

The authors would like to thank Richard Socher, Travis Oliphant, Charu Rudrakshi, Artem Harutyunyan, Iason Ofeidis, Diego Kiedanski, Vishnu Nair, Fayaz Rahman, Dyllan McCreary, Benjamin Hindman, Eduard Grigoryan, Kristina Grigoryan, Ben Chislett, Joubin Houshyar, Andrii Liubimov, Assaf Pinhasi, Vishnu Nair, Eshan Arora, Shashank Agarwal, Pawel Janowski, Kristina Arezina, Gevorg Karapetyan, Vigen Sahakyan and the open-source community including contributors. The project was funded by Activeloop. We also thank the CIDR reviewers for their feedback

REFERENCES

[1] 2006. Amazon S3. *GitHub* 2022, 1 (2006). https://aws.amazon.com/s3
[2] 2009. Clickhouse. *GitHub* 2022, 1 (2009). https://github.com/ClickHouse/ClickHouse
[3] 2010. Google Cloud Storage. *GitHub* 2022, 1 (2010). https://cloud.google.com/storage
[4] 2012. Google BigQuery. *GitHub* 2022, 1 (2012). https://cloud.google.com/bigquery
[5] 2014. Protocol Buffers - Google's data interchange format. *GitHub* 2022, 1 (2014). https://github.com/protocolbuffers/protobuf
[6] 2015. The Apache Software Foundation: Apache ORC. *GitHub* 2022, 1 (2015). https://github.com/apache/orc
[7] 2016. Feather. *GitHub* 2022, 1 (2016). https://github.com/wesm/feather
[8] 2016. Weaviate: The ML-first vector search engine. *GitHub* 2022, 1 (2016). https://github.com/semi-technologies/weaviate
[9] 2017. Apache Airflow. *GitHub* 2022, 1 (2017). http://airflow.incubator.apache.org
[10] 2017. The Apache Software Foundation: Apache Hudi. *GitHub* 2022, 1 (2017). https://hudi.apache.org
[11] 2017. CloudVolume: IO for Neuroglancer Datasets. *GitHub* 2022, 1 (2017). https://github.com/seung-lab/cloud-volume
[12] 2018. Amazon Athena. *GitHub* 2022, 1 (2018). https://aws.amazon.com/athena
[13] 2018. The Apache Software Foundation: Apache Arrow. *GitHub* 2022, 1 (2018). https://arrow.apache.org
[14] 2018. The Apache Software Foundation: Apache Hadoop. *GitHub* 2022, 1 (2018). https://hadoop.apache.org

[15] 2018. The Apache Software Foundation: Apache Iceberg. *GitHub* 2022, 1 (2018). https://iceberg.apache.org

[16] 2018. Feast: open source feature store for machine learning. *GitHub* 2022, 1 (2018). https://github.com/feast-dev/feast

[17] 2018. MinIO high performance object storage server compatible with Amazon S3 API. *GitHub* 2022, 1 (2018). https://github.com/minio/minio

[18] 2018. Petastorm. *GitHub* 2022, 1 (2018). https://github.com/uber/petastorm

[19] 2018. The WebDataset Format. *GitHub* 2022, 1 (2018). https://github.com/webdataset/webdataset

[20] 2019. The Apache Software Foundation: Apache Avro. *GitHub* 2019, 1 (2019). https://avro.apache.org

[21] 2019. LakeFS: data lake with Git-like repository. *GitHub* 2022, 1 (2019). https://github.com/treeverse/lakeFS

[22] 2020. Airbyte. *GitHub* 2022, 1 (2020). https://github.com/airbytehq/airbyte

[23] 2020. TensorStore: Library for reading and writing large multi-dimensional arrays. *GitHub* 2022, 1 (2020). https://github.com/google/tensorstore

[24] 2021. N5: specifies the primitive operations needed to store large chunked n-dimensional tensors, and arbitrary meta-data in a hierarchy of groups similar to HDF5. *GitHub* 2021, 1 (2021). https://github.com/saalfeldlab/n5

[25] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[26] Alex Aizman, Gavin Maltby, and Thomas Breuel. 2019. High performance I/O for large scale deep learning. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 5965–5967.

[27] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[28] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*.

[29] Alexei Baevski, Wei-Ning Hsu, Qiantong Xu, Arun Babu, Jiatao Gu, and Michael Auli. 2022. Data2vec: A general framework for self-supervised learning in speech, vision and language. *arXiv preprint arXiv:2202.03555* (2022).

[30] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[31] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax

[32] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[33] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[34] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[35] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

[36] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[37] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi (Eds.). OpenProceedings.org, 313–324. https://doi.org/10.5441/002/edbt.2019.28

[38] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[39] Andrew Ilyas Sam Park Hadi Salman Guillaume Leclerc, Logan Engstrom. 2021. The WebDataset Format. *GitHub* 2022, 1 (2021). https://github.com/libffcv/ffcv

[40] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

[41] Dong He, Supun Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *arXiv preprint arXiv:2203.01877* (2022).

[42] Yu Huang and Yue Chen. 2020. Survey of state-of-art autonomous driving technologies with deep learning. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 221–228.

[43] Tero Karras, Samuli Laine, and Timo Aila. 2019. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 4401–4410.

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[45] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 283–296.

[46] Ruslan Kuprieiev, skshetry, Dmitry Petrov, Paweł Redzyński, Peter Rowlands, Casper da Costa-Luis, Alexander Schepanovski, Ivan Shcheklein, Batuhan Taskaya, Gao, Jorge Orpinel, David de la Iglesia Castro, Fábio Santos, Aman Sharma, Dave Berenbaum, Zhanibek, Dani Hodovic, Nikita Kodenko, Andrew Grigorev, Earl, daniele, Nabanita Dash, George Vyshnya, maykulkarni, Max Hora, Vera, Sanidhya Mangal, and Wojciech Baranowski. 2022. *DVC: Data Version Control - Git for Data & Models*. https://doi.org/10.5281/zenodo.7039863

[47] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436.

[48] Kisuk Lee, Jonathan Zung, Peter Li, Viren Jain, and H Sebastian Seung. 2017. Superhuman accuracy on the SNEMI3D connectomics challenge. *arXiv preprint arXiv:1706.00120* (2017).

[49] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *European conference on computer vision*. Springer, 740–755.

[50] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, SangBin Cho, Eric Liang, and Ion Stoica. 2022. Exoshuffle: Large-Scale Shuffle at the Application Level. *arXiv preprint arXiv:2203.05072* (2022).

[51] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[52] Alistair Miles, John Kirkham, Martin Durant, James Bourbeau, Tarik Onalan, Joe Hamman, Zain Patel, shikharsg, Matthew Rocklin, raphael dussin, Vincent Schut, Elliott Sales de Andrade, Ryan Abernathey, Charles Noyes, sbalmer, pyup.io bot, Tommy Tran, Stephan Saalfeld, Justin Swaney, Josh Moore, Joe Jevnik, Jerome Kelleher, Jan Funke, George Sakkis, Chris Barnes, and Anderson Banihirwe. 2020. *zarr-developers/zarr-python: v2.4.0*. https://doi.org/10.5281/zenodo.3773450

[53] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging fAIg applications. In *13th fUSENIXg Symposium on Operating Systems Design and Implementation (fOSDIg 18)*. 561–577.

[54] Iason Ofeidis, Diego Kiedanski, and Leandros Tassiulas. 2022. An Overview of the Data-Loader Landscape: Comparative Performance Analysis. *arXiv preprint arXiv:2209.13705* (2022).

[55] Travis E Oliphant. 2006. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA.

[56] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. 2016. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499* (2016).

[57] 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.

[58] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[59] Masha Basmanova Kevin Wilfong Laith Sakka Krishna Pai Wei He Biswapesh Chattopadhyay Pedro Pedreira, Orri Erling. 2022. Velox: Meta's Unified Execution Engine. *Proceedings of the VLDB Endowment* (2022).

[60] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*. PMLR, 8748–8763.

[61] Pranav Rajpurkar, Jeremy Irvin, Kaylie Zhu, Brandon Yang, Hershel Mehta, Tony Duan, Daisy Ding, Aarti Bagul, Curtis Langlotz, Katie Shpanskaya, et al. 2017. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. *arXiv preprint arXiv:1711.05225* (2017).

[62] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-shot text-to-image generation. In *International Conference on Machine Learning*. PMLR, 8821–8831.

[63] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[64] Amit Sabne. 2020. Xla: Compiling machine learning for peak performance. (2020).

[65] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, Burcu Karagol Ayan, S Sara Mahdavi, Rapha Gontijo Lopes, et al. 2022. Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding. *arXiv preprint arXiv:2205.11487* (2022).

[66] Tom van Bussel Samwel, Herman van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. (2022).

[67] Christoph Schuhmann, Romain Beaumont, Cade W Gordon, Ross Wightman, Theo Coombes, Aarush Katta, Clayton Mullis, Patrick Schramowski, Srivatsa R Kundurthy, Katherine Crowson, et al. 2022. LAION-5B: An open large-scale dataset for training next generation image-text models. (2022).

[68] Christoph Schuhmann, Richard Vencu, Romain Beaumont, Robert Kaczmarczyk, Clayton Mullis, Aarush Katta, Theo Coombes, Jenia Jitsev, and Aran Komatsuzaki. 2021. Laion-400m: Open dataset of clip-filtered 400 million image-text pairs. *arXiv preprint arXiv:2111.02114* (2021).

[69] Philip Schwan et al. 2003. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, Vol. 2003. 380–386.

[70] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.

[71] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. 2010. The hadoop distributed file system.. In *MSST*, Vol. 10. 1–10.

[72] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144.

[73] K Stumpf, S Bedratiuk, and O Cirit. 2018. Michelangelo PyML: introducing Uber's platform for rapid python ML model development. *Uber. See: https://eng. uber. com/michelangelo-pyml* (2018).

[74] Pedro Javier Ortiz Suárez, Benoît Sagot, and Laurent Romary. 2019. Asynchronous pipeline for processing huge corpora on medium to low resource infrastructures. In *7th Workshop on the Challenges in the Management of Large Corpora (CMLC-7)*. Leibniz-Institut für Deutsche Sprache.

[75] Squirrel Developer Team. 2022. Squirrel: A Python library that enables ML teams to share, load, and transform data in a collaborative, flexible, and efficient way. *GitHub. Note: https://github.com/merantix-momentum/squirrel-core* (2022). https://doi.org/10.5281/zenodo.6418280

[76] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive-a petabyte scale data warehouse using hadoop. In *2010 IEEE 26th international conference on data engineering (ICDE 2010)*. IEEE, 996–1005.

[77] Kathryn Tunyasuvunakool, Jonas Adler, Zachary Wu, Tim Green, Michal Zielinski, Augustin Žídek, Alex Bridgland, Andrew Cowie, Clemens Meyer, Agata Laydon, et al. 2021. Highly accurate protein structure prediction for the human proteome. *Nature* 596, 7873 (2021), 590–596.

[78] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[79] Deepak Vohra. 2016. Apache parquet. In *Practical Hadoop Ecosystem*. Springer, 325–335.

[80] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.

[81] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems* 32 (2019).

[82] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[83] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*. 649–657.