Studying Storage system backends for ML Workloads

Group 29 - Preetham Srinivasa Kikkeri, Ranjitha Kosgi, Varun Kaundinya

1 Introduction

With increasing compute power of GPUs, dataset loading and pre-processing is going to be a significant issue. As accelerators are getting faster, the storage media and data buses feeding the data have not kept pace [14] and the ever-increasing size of training data further compounds the problem. Loading many batches sampled randomly is the common workload for deep learning (DL). For example, the SGD model optimizer requires loading a mini-batch of training samples in a randomized order for every iteration. This is important for accelerating the model's convergence speed and decreasing the noise learned from the input sequence. However, such iterative loading of samples imposes nontrivial I/O pressure on storage systems which are typically designed and optimized for large sequential I/O [12, 13]. Deep Learning system architects strive to design a balanced system where the computational accelerator – FPGA, GPU, etc, is not starved for data. However, feeding training data fast enough to effectively keep the accelerator utilization high is difficult. Further, different ML workloads have different storage requirements. NLP models store texts, image models need images, embedding-based models need numerical arrays, etc. Each of these has different compute and disk storage bandwidth requirements. A storage system that might work well for a particular workload might perform poorly with a different workload.

In this project, we try to benchmark the performance of 2 storage systems - MinIO and HDF5, for different ML models and reason about the model performance on said storage systems by comparing them with the Ext4 as the baseline, and using different PyTorch dataloader configurations. Another important aspect of this project is to customize the PyTorch datasets to have complete control over the training data such as accessing the data and loading the required data into memory.

2 Background

2.1 Storage Systems

The storage systems used in our experiments are described below:

Ext4 - Ext4 is a file system used on Linux and other Unix-like operating systems. It is designed to support very large files and file systems, and it includes features such as journaling and checksumming to improve reliability and robustness. It is the default file system for many Linux distributions and is also used on other operating systems and embedded systems. This file system is used as the baseline to compare other systems.

MinIO [1] - MinIO is an open-source object storage server that stores and retrieves data as objects. It is a high-performance system with the ability to scale at a competitive cost, and handle unstructured data such as photos, videos, and log files. It can be deployed on a single machine or in a distributed configuration and can be accessed via a web interface or using various client libraries. In addition, it can be configured with a variety of storage backends such as Local Disk, Network Attached Storage (NAS), Cloud Storage Offerings. MinIO supports Amazon S3 protocol, making it easier to integrate with other systems that support S3. Legacy storage solutions won't suffice for today's artificial intelligence and machine learning (AI/ML) workloads, as they are not linearly scalable and cannot handle huge pool of unstructured data. This is where object storage shines - the storage type that is durable, available and can scale limitlessly to tens of petabytes and beyond within a single, global namespace. These advantages motivated us to benchmark MinIO.

HDF5 [2]- The Hierarchical Data Format version 5 (HDF5), is an open-source file format for storing and organizing large amounts of data in a hierarchical structure. It is designed to handle data that is too large to fit in memory, as well as to allow fast access to specific parts of the data. It is a self-describing file format, which means that it stores not only the data, but also metadata describing the data and the structure of the file like data type, shape, and dimensions of the data. It uses a hierarchical file format, where it stores data in a tree-like structure with multiple levels of nesting. Each level in the hierarchy is called a ”group”, and groups can contain datasets (arrays of data) or other groups. This allows HDF5 to represent complex data structures, such as multi-dimensional arrays with metadata and annotations. HDF5 is efficient and fast, both in terms of file size and access time. It supports a variety of data types and can store data in com-

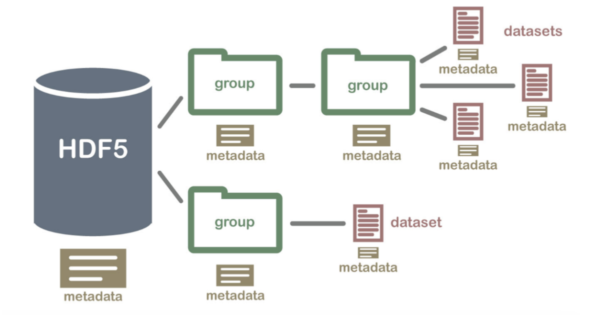

Figure 1: HDF5 File Structure

pressed or uncompressed form. It also supports parallel I/O, which allows multiple processes to read and write to the same file simultaneously. HDF5 is supported by many programming languages and tools, including open-source languages like R and Python and open GIS tools like QGIS, which makes it a suitable choice for our experiments.

2.2 Machine Learning Models

The Machine Learning models chosen for the evaluation of the storage systems are *BERT* and *ResNet*. The reasoning behind them was to have models reading data from different file formats and belonging to completely different domains with varying data and access requirements.

BERT (Bidirectional Encoder Representations from Transformers) is a type of transformer-based neural network architecture designed specifically for natural language processing (NLP) tasks. It was developed by researchers at Google and introduced in a paper published in 2018 [15]. The key innovation of BERT is its use of bidirectional attention, which allows the model to take into account the context of a word in both the left and right sides of a sentence. This is in contrast to traditional language models, which only consider the context to the left of a word. BERT is trained on a large dataset of unlabeled text and is able to learn the structure and patterns of language by predicting missing words (also known as ”masked language modeling”). This allows it to capture the context of a word within a sentence, as well as its relationship with other words in the sentence. BERT has achieved state-of-the-art results on a wide range of NLP tasks, including text classification, language translation, and question answering. The BERT model used in our experiments completes missing words in masked Italian sentences [19].

ResNet (Residual Network) is a type of convolutional neural network (CNN) designed to enable the training of very deep networks. It was introduced in a paper published by researchers at Microsoft in 2015 and has since become a widely used architecture for image classification and other computer vision tasks [16]. ResNet addresses the infamous vanishing gradient problem by introducing skip connections, which allow the gradient to bypass one or more layers and be directly backpropagated to shallower layers. This helps to stabilize the training process and allows for the training of much deeper networks. ResNet has achieved state-of-the-art results on a number of image classification benchmarks and has been widely adopted in many computer vision applications. In our model we use resent model to classify a document dataset. The idea is to assign a target label identifying the type of document (scientific, newspaper, art) based on the contents of the document.

A Dataloader is used to feed the data to the ML models. It is a utility that is used to load data from a dataset and feed it to a model during training. It supports various other features, such as loading data in parallel using multiple worker processes, applying transformations to the data on the fly, and handling missing or incomplete data. It is a convenient and powerful tool for loading and processing data. It uses an iterative object style that allows the data to be processed in a lazy and memory-efficient manner, as the data is only loaded and processed as needed. In this project, the *PyTorch DataLoader* is used to load data from the backend storage systems. The different parameters with respect to batching, transforming, shuffling, and the number of workers are tuned to benchmark MinIO and HDF5. The PyTorch dataloader works by calling the *__getitem__* method of the underlying dataset object to load the instance of the data corresponding to the specified index. Based on the shuffle parameter the data accessed by the index can be sequential (shuffle = False) or it can be random (shuffle = True), where the index is limited by the *__len__* method of the dataset which specifies the upper limit.

3 Related Work

Potential bottlenecks to store large collections of small files have been pointed out across the file system community [9, 10], but none of these studies specifically considered the problem in the context of deep learning. Also, lot of studies [3, 4, 5, 6] have documented the evaluation of different Deep Learning applications on different HPC systems, but all of these mainly deal with the computation characterization. [17] evaluated how various hardware configurations affect the overall performance of deep learning, including storage devices, main memory, CPU, and GPU. They compared training time using four storage devices: single disk, single SSD, Disk Array, and SSD Array.

[8]Explores the impact of different parameters, including storage location, storage disaggregation granularity, access pattern, and dataformat. However, the optimal storage system is a combination of the backend storage system and type of workload., [7] evaluates image storage systems for training deep neural networks, including key-value databases, file systems, and in-memory Python/C++ array. This work also explores the use of HDF5 as one of the storage formats for images. Dataloaders can impact the the training throughput. Efforts have been made previously to benchmark the performances of different dataloaders - [11] benchmarks the performances of the current state-of-the dataloaders for 2 different learning tasks: CIFAR-10 for image classification, and CoCo for object detection.

Our work focuses on studying the impact of underlying storage systems on the ML throughput and aims to benchmark storage systems (MinIO and HDF5) for different ML workloads which deal with various data types such as images, text.

4 Performance Evaluation

4.1 Experimental Setup

The models are trained on a single GPU node with the following configuration:

Machine Configuration: Architecture: x86_64; GPU(s): NVIDIA T4; CPU(s): 4; Thread(s) per core: 2; Core(s) per socket: 4; Socket(s): 1

Versions: CUDA11.3.GPU, PyTorch 1.12.1

4.2 ResNet

The Resent Model is trained using the document dataset by IBM called PubLayNet[18] containing 335,703 training images and 11,245 validation images. This document image dataset corresponds to approximately 80GB of storage space. The model assigns a target label identifying the type of document content(scientific, newspaper, art) based on the contents of the document. The dataset also had some classic transformations to build invariance (flipping, resizing, blurring). We trained ResNet for 80 iterations using all the three storage backends - Ext4, MinIO, HDF5 with batch size of 8.

4.2.1 Approach 1

In the first approach, the data loader is fed a custom dataset that provides the images lazily. Here the file names of all images are stored in memory and the respective images are lazily loaded when the corresponding filename index is accessed by the data loader. The images are accessed in random order (shuffle = True). The baseline performance is the ext4 file system. Iteration times for ext4 can be seen in the figure 2. It can be seen that dataloading consumes a significant fraction (30%) of the iteration time. The average data loading time is 298ms and the average training time is 665ms.

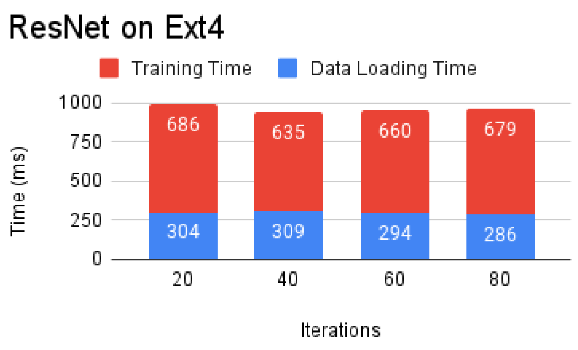

Figure 2: Approach 1 in Ext4

MinIO: With MinIO, each image is stored as a separate object in its object store, Like in Ext4, the subsequent MinIO object consisting of the image is lazily loaded when the corresponding index is accessed by the data loader.

HDF5: There are different ways of storing & structuring in HDF5. The hierarchy chosen was to store each image as a separate dataset. The image is converted to binary format before storing it in HDF5, this is the most efficient format to store images in HDF5 as HDF5 is extremely inefficient in storing NumPy arrays (more on this in section 4.2.2). Like in Ext4, the subsequent HDF5 dataset consisting of the image is lazy loaded when the corresponding index is accessed by the dataloader.

We can see from figure 3 that Ext4 and HDF5 have a similar performance while MinIO has a slightly higher load time. The slightly higher load time on MinIO is due to the extra API interaction to load the data from the MinIO store.

The above experiment doesn't utilize any of the key features in the storage systems and as expected the performance from these are similar or worse than ext4.

4.2.2 Approach 2 - Range Fetch

In the second Experiment, the data loader and dataset was tweaked to return multiple elements per index call, so 1 data index corresponds to #batch_size number of im-

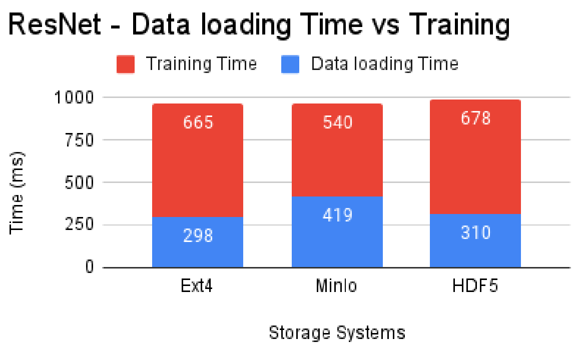

Figure 3: Approach 1

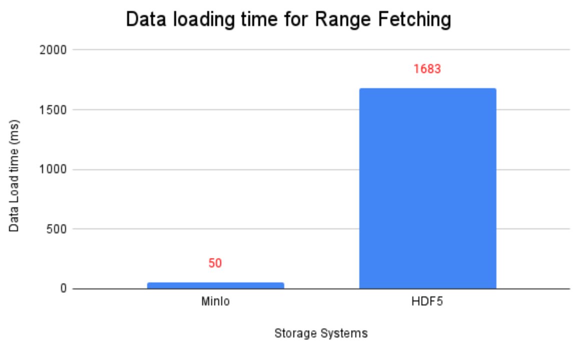

Figure 4: Approach 2 - Range fetch

ages. This allowed capturing intrinsic data loading features from MiniIO and HDF5.

MinIO: For MinIO, unlike approach 1, multiple images are stored per object. Since the API specification is granular on the object level meaning that there would be a reduction in IO calls to retrieve images from the system. In this setup #batch_size images (8 images) are stored per object. This means, to load a batch of images to the model, 1/#batch_size IO calls are required.

Dataloading time decreased to *50 ms* (refer figure 3). This is a significant improvement as compared to the previous dataloading time of 419ms in experiment 1. This is because we are able to get 8 images in a single IO operation as compared to 8 IOs to read 8 image objects previosuly.

HDF5: For HDF5, all the images are stored in a single dataset. Due to restrictions in HDF5, storing the images in this hierarchy would mean storing images as NumPy arrays (as opposed to binary in experiment 1). This results in HDF5 taking 10-20x the storage space as compared to experiment 1, which lead to disk space issues. To avoid this, compressed versions of the images are stored in the dataset. Refer section 4.2.3 for details on compression.

HDF5 provides an option of *range reading*, allowing selective reading of a range of data within a dataset in a single IO. All the images are stored in a single dataset, unlike approach 1 where each image is stored as a dataset. This allowed reading multiple images in a single IO, reducing the overall IO calls from #batch_size to 1/#batch_size.

The data loading time became significantly worse after compressing - from *310 ms to 1683 ms* (refer figure3). This is due to the overhead in decompressing the image data during each load.

4.2.3 Storage Footprint

Choosing the right storage system depends on the model we are training and the type of data that the model uses. Dataloading time as well as storage footprint need to be taken into account to optimize the performance. This is particularly important when it comes to HDF5, as it supports different data formats and hierarchies.

HDF5 files take large amount of space for storing image data as NumPy arrays. The file size increases rapidly with the number of images and can go upto 100X the size of the original images. Images stored as numpy arrays in HDF5 took 3329KB per image data as compared to the original image size of 255KB. Surprisingly, The data loading time is 220 ms which is better than all the other storage systems. HDF5 performs well for low-latency random accesses but it is not optimized for storage space.

One alternative to this is storing the images in binary format. In this approach, the size of images stored in binary format in HDF5 as well as the data loading time are comparable to that of Ext4, as can be seen in figure 5

Another alternative is to opt for compression in HDF5. The python package h5py supports a few compression filters such as GZIP, LZF, and SZIP. When one of the compression filters is used, data will be compressed on its way to the disk and will be decompressed when read from it. In our approach, GZIP compression is used with a compression factor of 4. An image of size 255KB when stored in HDF5 without compression took *3.3MB* of space while it took *247KB* with compression. By using compression like GZIP, the memory footprint is similar to that of Ext4 which can been seen in figure 5. However, the dataloading time shoots up to *30s*. This is due to the overhead in decompressing the image data during each load. So, compression is a viable option when storage requirements are a bottleneck. If dataload times need

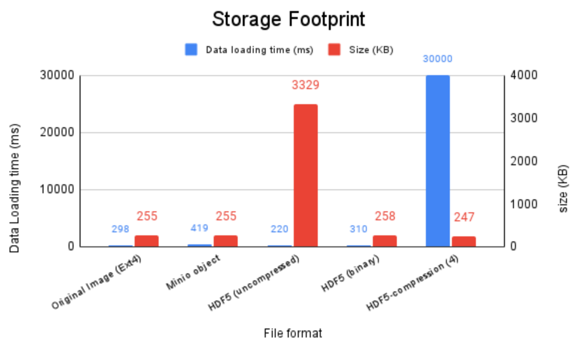

Figure 5: Approach 1 across storage systems

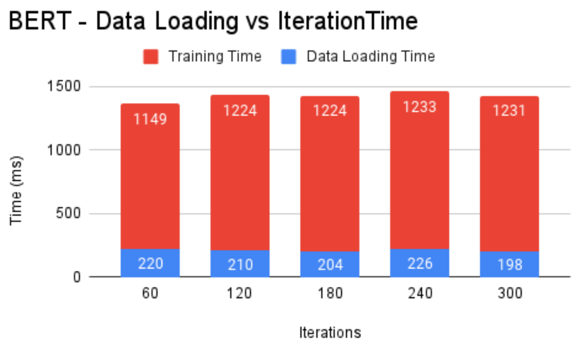

Figure 6: BERT-Ext4

to be optimized, then it is better to use the default option of numpy arrays or binary data in HDF5.

4.3 BERT

The BERT is trained using the *huggingface oscar unshuffled_deduplicated_it* dataset. Couple of preprocessing steps are performed on the raw data. Firstly, this dataset is converted into .txt files, by saving 10,000 lines in each file. This generates 3000 text files with a cumulative storage footprint of approximately 30GB. Next, these text files are passed through the tokenizer to map these tokens to Ids, and manages special tokens like the End of line, Beginning of sequence, Padding token, etc. A Custom dataset is written to read lines from text files and convert each line into an object consisting of 3 tensors, *input_ids* — ids representing the words and masked words, *attention_mask* — a tensor of 1s and 0s, marking the position of 'real' tokens/padding tokens (used in attention calculations), *labels* — token_ids with no masking (target). The model completes missing words in masked Italian sentences or phrases.

BERT is trained for 300 iterations using all three storage backends - Ext4, MinIO, HDF5 with batch sizes as 8,16,20 and with different access patterns such as reading the lines sequentially vs random accesses using shuffling.

4.3.1 Approach 1 - Sequential loading

In the first experiment, the lines are read sequentially (shuffle = false). The data loader is fed a custom dataset that maintains 2 pointers one tracking the index of the text files from an array of all the text file names and a second pointer to track the line in a particular opened text file. This 2 level pointer system allows tracking the input dataset granularly. Everytime a new file is opened the first pointer is incremented and all the lines in the opened files are processed and saved in memory. Subsequent calls for data are satisfied by the 2nd pointer traversing the in-memory list. This approach mirrors state-of-the-art setups where RAM is used as a temporary buffer to reduce latency and IO overhead. In order to open new files on regular intervals (for experiment purposes), only 1000 lines from the opened file are saved in memory.

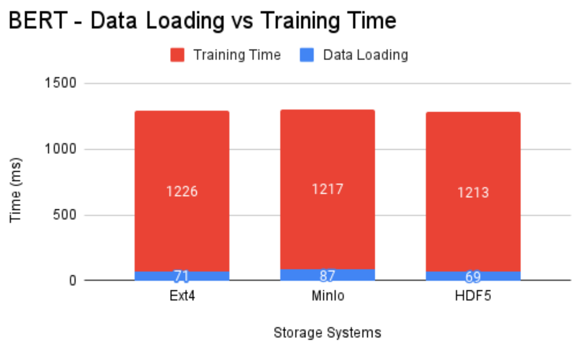

Figure 7: BERT - Approach 1

Iteration times for NLP with Ext4 storage backend can be seen in figure 6. The data loading time makes up *15%* of iteration time. This is significantly lower than the images in the Resnet model, the reason being image files are typically larger in size. Another reason being IO call to load data happening once every 1000 lines are processed. The latency of reading from memory is significantly lower than IO (until we move onto the next text file in our dataset), hence the average is smaller.

To capture metrics for HDF5, each text file is stored as a separate dataset and the same 2-pointer system is used to track the dataset and process lines as done in Ext4. In MinIO, each text file is stored as an object and the same setup is used as the other 2. All the storage systems show a *similar performance* using this approach.

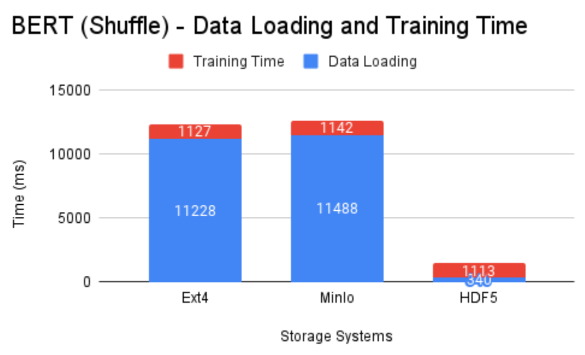

Figure 8: BERT - Approach 2 (Shuffle)

4.3.2 Approach 2 - Shuffle read

Shuffling the data can be useful in training a machine learning model, because it helps to prevent the model from overfitting to the order of the data. By presenting the data in a different order at each epoch, the model is forced to learn the underlying patterns in the data rather than just memorizing the order of the examples.

The shuffle parameter in a PyTorch DataLoader determines whether the data should be shuffled at the beginning of each epoch. If shuffle=True, the data will be shuffled randomly before the first batch is returned. If shuffle=False, the data will be returned in the order it appears in the dataset.

In this approach, HDF5 outperforms the rest by a huge margin, as seen in figure 8. This is because each time a batch of training data is fetched (eg: 16 lines if batch size is 16), each of these lines are read randomly from any of the text files. In the worst case scenario, 16 text files need to be loaded into memory even though only one line is read from each of those files. This adds a significant IO overhead as well as memory overhead for Ext4 and MinIo. On the other hand, HDF5's range reading allows reading a range of data within a dataset without having to load the whole dataset in memory. So a line can be read selectively without loading the whole text file.

5 Structuring in HDF5

There are many ways to structure an HDF5 file, depending on the needs of an application. For example, one might have a file hierarchy with several groups, each containing multiple datasets, or a simple file hierarchy with just a few datasets at the top level. The choice of file hierarchy structure will depend on the requirements of the machine learning model, access patterns associated with the model and the type of data stored.

Here, experiment was carried out to store the text data in different ways. A single group is used with different structures for datasets within the group.

1. One text file per dataset - Each text file (consisting of 10000 lines) is stored as a separate dataset.
2. All lines in a single dataset - Lines from all the text files are put together into a single dataset.
3. One line per dataset - Each line is stored as a separate dataset.

5.1 Range Fetching in HDF5

Dataloading time for reading a contiguous section of 10000 lines was measured, for each of the above mentioned ways to structure data. It can be see from figure 9 that storing all the lines stored as part of a single dataset gives the best performance. This is because HDF5 allows reading ranges of data within a dataset in a single IO operation. Storing each text file in a dataset performs slightly worse. This is mostly due to a 2-level lookup for first finding the corresponding dataset and then reading the range of lines. Lastly, storing one line per dataset has poor performance. This is because we can only load one dataset at a time, hence requiring a large number of IO operations.

5.2 Randomised fetching in HDF5

In this experiment, dataloading time was measured for reading one data item at a time randomly for a batch of 16. This activity was repeated for each of the above-mentioned ways to structure data. It can be seen from the figure 9 that storing each text file in a dataset performs poorly. This is due to a 2-level lookup, first for finding the corresponding dataset and then reading the corresponding line, which is repeated for each data item fetch. Storing all lines in a single dataset performs better. This is because a single dataset contains all the data and each data item can be fetched with the index, leading to O(1) time complexity. Storing each item as a dataset performs a little worse. This is due to the fact that HDF5 takes linear time to search for a dataset with a key in a group, that is O(n).

6 DataLoader - Tuning Observation

1. **number of workers :** When the number of workers were increased, the dataloading time dropped significantly due to pre-fetching. But, beyond 2 workers, there was no effect on performance.

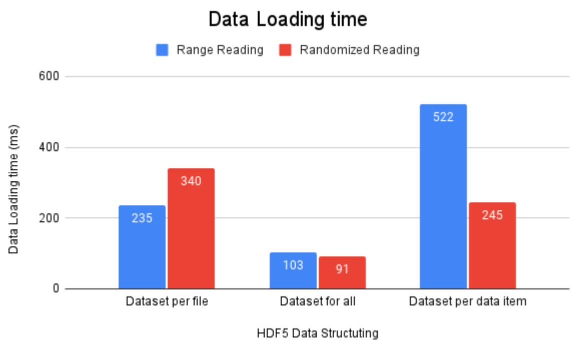

Figure 9: HDF5 structuring and access patterns

2. **number of threads :** When this is increased, we see that there is 10-15% decrease in data loading time. This is predominantly due to the parallel processing of transformation once loaded by the dataset.

3. **batch size :** For all the experiments performed there was a linear correlation (increase/decrease) based on trends when the batch size was increased or decreased. A batch size of 32 lead to a CUDA OUT OF MEMORY issue, due to the memory constraints of the setup.

7 Conclusion

In this project, we benchmark the performance of two storage systems - MinIo and HDF5, for machine learning models - Resnet and BERT over a single-node GPU. To configure and run efficient Deep Learning training piplines, one must choose the right storage systems, data loader configuration and data formats while keeping in mind the training methodology of the model, access patterns and resource constraints such as compute, memory and disk space. The synchrony among these attributes play significant role in bringing down the training time. With HDF5 we found the way of structuring data (groups and datasets) along with different storage formats (compression, binary, NumPy arrays) leading to differing latency and storage footprints. HDF5 suits best when memory is a bottleneck since it allows us to selectively read data. MinIO being more of a generic object storage solution isn't catered for any specific kind of workloads or datasets. The key advantage provided by MinIO is the abstraction of hiding the data as objects which gives us the flexibility to customize MinIO for different workloads. To get the best outcomes, one must balance model requirements, input data format and latency requirements to choose the right storage solutions.

8 Contributions

- All the members were involved in designing the experiments, set up and configuration of the experiments on cloud.
- Preetham - focused on HDF5 related exploration.
- Ranjitha - on MinIO exploration.
- Varun - on Deep learning models.

9 Acknowledgements

We would like to extend our Thanks to Professor Shivaram Venkataraman, Roger Waleffe and Saurabh Agarwal for all the support throughout this project.

References

[1] MinIO *: https://min.io/*

[2] HDF5 *:https://www.hdfgroup.org/HDF5/*

[3] Jiazhen Gu, Huan Liu, Yangfan Zhou, and Xin Wang. 2017. *Deepprof: Performance analysis for deep learning applications via mining gpu execution patterns*. arXiv preprint arXiv:1707.03750 (2017).

[4] Mauricio Guignard, Marcelo Schild, Carlos S Bederián, Nicolás Wolovick, and Augusto J Vega. 2018. *Performance characterization of state-of-the-art deep learning workloads on an ibm" minsky" platform*. In Proceedings of the 51st Hawaii International Conference on System Sciences.

[5] Yuriy Kochura, Sergii Stirenko, Oleg Alienin, Michail Novotarskiy, and Yuri Gordienko. 2017. *Performance analysis of open source machine learning frameworks for various parameters in single-threaded and multi-threaded modes*. In Conference on computer science and information technologies. Springer, 243–256.

[6] Xiaqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, and Weimin Zheng. 2016. *Performance analysis of gpu-based convolutional neural networks*. In 2016 45th International Conference on Parallel Processing (ICPP). IEEE, 67–76

[7] Seung-Hwan Lim, Steven R Young, and Robert M Patton. 2016. *An analysis of image storage systems for scalable training of deep neural networks*. system 5, 7 (2016), 11.

[8] Peng Cheng, Haryadi S. Gunawi. 2021. *Storage Benchmarking with Deep Learning Workloads*. In

the Seventh workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware (in conjunction with ASPLOS'16) conference.

[9] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. 2010. *Finding a Needle in Haystack: Facebook's Photo Storage*. In OSDI, Vol. 10. 1–8.

[10] Kai Ren and Garth Gibson. 2013. *TABLEFS: enhancing metadata efficiency in the local file system*. In Proceedings of the 2013 USENIX conference on Annual Technical Conference. 145–156.

[11] Iason Ofeidis, Diego Kiedanski, Leandros Tassiulas. 2022. *An Overview of the Data-Loader Landscape: Comparative Performance Analysis*

[12] Timothy Prickett Morgan. 2018. Removing The Storage Bottleneck for AI. www.nextplatform.com/2018/03/29/removing-the-storage-bottleneck-for-ai

[13] Open Data Science. 2018. Data Storage Keeping Pace for AI and Deep Learning. https://medium.com/predict/data-storage-keeping-pace-for-ai-and-deeplearning-ad3e75e1c67a

[14] https://www.nextplatform.com/2018/03/29/removing-the-storage-bottleneck-for-ai/

[15] Devlin, J., Chang, M., Lee, K., Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv. https://doi.org/10.48550/arXiv.1810.04805

[16] He, K., Zhang, X., Ren, S., Sun, J. (2015). Deep Residual Learning for Image Recognition. arXiv. https://doi.org/10.48550/arXiv.1512.03385

[17] Xiaqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, and Weimin Zheng. 2016. Performance analysis of gpu-based convolutional neural networks. In 2016 45th International Conference on Parallel Processing (ICPP). IEEE, 67–76.

[18] https://developer.ibm.com/exchanges/data/all/publaynet/

[19] https://towardsdatascience.com/how-to-train-a-bert-model-from-scratch-72cfce554fc6