OneAccess: Concurrent Data Access Layer for Analytics

Aarati Kakaraparthy, Abhay Venkatesh, and Srujith Poondla

1 Introduction

ML sampling and training systems are typically disparate in practice. Usually, ML practitioners write their own data loaders for each dataset, which are often not optimized. Frameworks come with limited functionality data loaders that provide basic features, but are not feature rich as they can be. Furthermore, training and data loading are compute tasks that are isolated from each other. We can make significant performance improvements if we can introduce synergy between training system and data sampling system. Machine learning systems could benefit from concurrency that composes data loading and compute with each other.

With the access to specialized hardware like TPU [1] and GPU, computation time has reduced while data access time has not seen the same order of improvement. Machine learning frameworks like PyTorch attempt to overcome this bottleneck by performing data loading concurrently through multiple processes. Randomness of data is crucial for machine learning algorithms, which in turn means that data loading performed can lead to requests at any arbitrary location of the memory hierarchy depending on where the sampled data is located. To overcome this limitation of data access in machine learning frameworks, we have designed OneAccess.

Our goal is to provide ML practioners with a one-stop-shop for all data loading. In this report, we describe the background, design, and implementation of the first version of such a system that we called *OneAccess*. The three main ideas behind the design of OneAccess are:

- Utilizing reservoir sampling for loading data can bring significant efficiency gains through sequential access at all levels of the storage hierarchy. We have obtained gains as high as 3.6x in our experiments.
- Reservoir sampling allows us to repeatedly sample over any memory hierarchy for maximum flexibility of data infrastructure.
- Through reservoir sampling, we can retain guarantees of perfect randomness, while performing sequential accesses at the same time.

2 Background

In this section, we give some background on Machine Learning (Section 2.1) and few experiments to highlight the bottleneck of data loading (Section **??**). Finally, we describe reservoir sampling and prove some of its important properties (Section 2.3).

2.1 Machine Learning

The main goal of machine learning is to fit a model to approximate some function using data representative of that function. The three main pieces in a typical supervised learning setting are:

- Data: $((x_1, y_1), ..., (x_n, y_n))$ drawn from input data domain $x \sim X$, and response variable domain $y \sim Y$.
- Model: logistic regression, neural network, support vector machine are some examples of models that are commonly utilized.
- Fitting procedure: comprised of objective function (with constraints optionally), and optimization algorithm. An example could be, a deep learning model with cross-entropy loss function and uses stochastic gradient descent to minimize that loss.

ResNets, short for Residual Networks, were first introduced by He et. al. in their 2015 paper [4]. ResNets are deep convolutional neural networks comprising of many blocks that look like Figure 1. LeCun et. al. give a great overview of how convolutional neural networks (CNNs) work in their 2015 paper ([6]). CNNs are built with blocks that look like Figure **??**.

We use ResNets for our experiments because they achieve cutting edge performance in many important machine learning tasks, and at the same time we can easily vary the size of ResNets to analyze performance of our system.

2.2 Compute vs Data Loading

We perform some tests on Cifar-10 and MS COCO using the PyTorch data loader with the ResNet-18 and ResNet-152 models. This is done so as to give a span of anal-

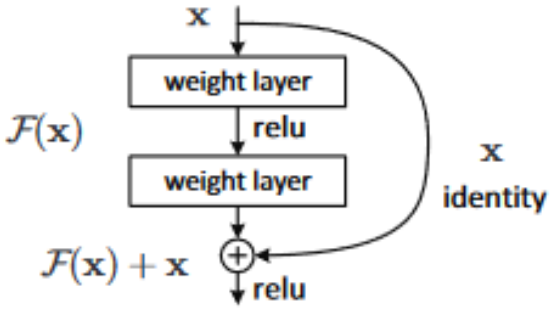

Figure 1: Residual Learning: a building block.

Type	Mean(s)	Median(s)	Standard Deviation(s)
Data Time	0.01205	0.01560	0.01560
Compute Time	0.01560	0.04683	0.05001

Table 1: Statistics for Cifar-10 with ResNet-18 model

ysis across dataset size and model size. The CIFAR-10 dataset is of size 187MB, and consists of 60k 32x32 colour images in 10 classes [5]. MS COCO, on the other hand is a much larger dataset with 118K images and of size 18GB [7]. We run ResNet-18 (smaller) and ResNet-152 (comparitively larger) models on both these datasets.

The experiments for Cifar-10 were run on a machine with Intel Core i7 2.4GHz processor, 16GB of RAM, and an Nvidia GTX 1070 GPU with 8GB memory. Experiments on MS-COCO dataset were run on a higher end machine with Intel Core i7 3.6GHz processor, 48GB of RAM, Nvidia GTX 1080 Titan GPU with 11GB of memory. Tables 1 and 2, and Figures 2 and 3 show results for Cifar-10 dataset. The results for MS-COCO are present in Tables 3 and 4, and shown in Figures 4 and 5.

Our experiments show that since the images are tiny, data loading does not particularly present a bottleneck for the Cifar-10 dataset on both the models. However, for MS-COCO we observe that data loading is a bottleneck

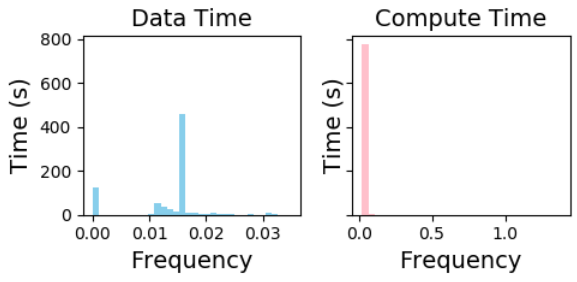

Figure 2: Data and compute times for Cifar10 on ResNet-18

Type	Mean(s)	Median(s)	Standard Deviation(s)
Data Time	0.00915	0.01561	0.00766
Compute Time	0.61072	0.60923	0.06447

Table 2: Statistics for Cifar-10 with ResNet-152 model

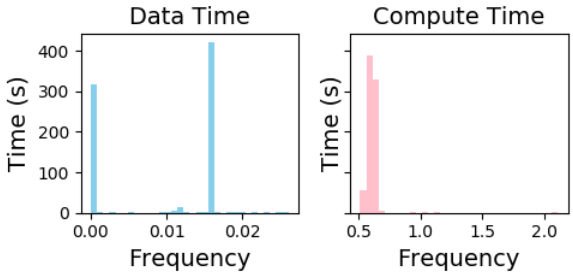

Figure 3: Data and compute times for Cifar10 on ResNet-152

for Resnet-18, which is a smaller model among the two.s

2.3 Reservoir Sampling

We perform reservoir sampling across memory hierarchy to

1. leverage sequential sampling to minimize random access costs
2. minimize access costs pertaining to data movement across memory hierarchy

Theorem 1. *All elements have equal probability of being sampled.*

Consider a dataset $D = \{d_1, ..., d_N\}$ of size N. We are interested in generating a sample of size $m \leq N$ using reservoir sampling. Then, each item in the sample has a probability of $\frac{m}{N}$ of being kept in the reservoir.

Proof. We split our dataset into two parts:

$$\{d_1, ..., d_m\}$$

$$\{d_{m+1}, ..., d_N\}$$

The probability that each $d_i \in \{d_1, ..., d_m\}$ remains is the probability that d_i is not ejected consecutively for iterations $j \in \{m+1, ..., N\}$. The probability that d_i is

Type	Mean(s)	Median(s)	Standard Deviation(s)
Data Time	0.37531	0.37521	0.01106
Compute Time	0.04049	0.03983	0.00217

Table 3: Statistics for MS COCO with ResNet-18

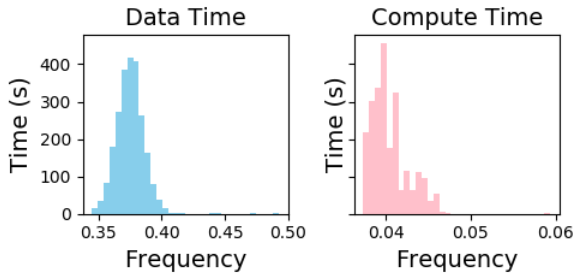

Figure 4: Data and compute times for MS COCO on ResNet-18

Type	Mean(s)	Median(s)	Standard Deviation(s)
Data Time	0.36517	0.35797	0.03396
Compute Time	0.71945	0.71787	0.03299

Table 4: Statistics for MS COCO with ResNet-152

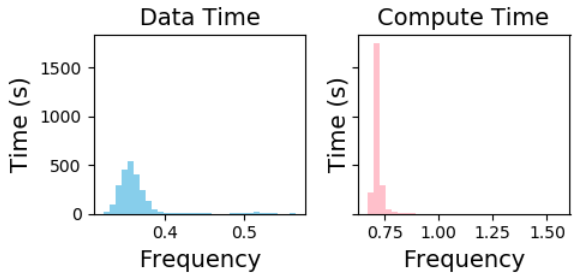

Figure 5: Data and compute times for MS COCO on ResNet-152

Data: dataset of size N $\geq$ m
Result: filled reservoir R
create reservoir R of size m ;
Read the first m items and fill the reservoir;
for *each additional item k (m + k items total)* **do**
  randomly select integer $s \in [0, m+k)$;
  if $s < m$ **then**
    put item in slot s;
  else
    drop the item;
  end
end

Algorithm 1: Reservoir Sampling

not ejected at iteration j is $1 - \frac{1}{j}$ since d_i is ejected with probability $\frac{1}{j}$. Therefore, the probability that d_i remains at the end is:

$$\prod_{j=m+1}^{N} 1 - \frac{1}{j} = (1 - \frac{1}{1+m})(1 - \frac{1}{2+m})...(1 - \frac{1}{N})$$

$$= (\frac{m}{m+1})(\frac{m+1}{m+2})...(\frac{N-1}{N}) = \frac{m}{N}$$

The probability that each $d_i \in \{d_{m+1}, ..., d_N\}$ remains in the reservoir is the probability that d_i is selected and that it is not ejected subsequently. Each element d_i is selected with probability $\frac{m}{j}$ for iterations $j \in \{m+1, ..., N\}$. If selected at iteration j, it is subsequently not ejected at iterations $k \in \{j+1, ..., N\}$ with probability

$$\prod_{k=j+1}^{N} 1 - \frac{1}{k} = \frac{j}{N}.$$

Therefore, the probability that d_i is kept by the end is

$$\frac{j}{N}\frac{m}{j} = \frac{m}{N}.$$

□

Theorem 2. *All samples have an equal probability of being chosen.*

Consider a dataset of size N. We are interested in generating a sample of size $m \leq N$ using reservoir sampling. Then, each such sample has an equal probability of $\frac{1}{{}^{N}C_{m}}$ being selected.

Proof. Final sample consists of elements $\{a_1, a_2, ..., a_m\}$, where $\forall i \leq j, a_i \leq m$ and $\forall i > j, a_i > m$. In other words, a subset of the elements ($\{a_1, ..., a_j\}$) of the initial sample are retained in the final sample.

We calculate the probability contribution p_k from all the elements.

- Elements $k \in \{m+1, ..., a_{j+1} - 1\}$: These elements, if chosen, cannot replace the existing j elements in the reservoir. Their contribution is $\frac{k-m}{k} + \frac{m}{k} \times \frac{m-j}{m} = \frac{k-j}{k}$.
- Elements $\forall i > j, k \in \{a_i + 1, ..., a_{i+1} - 1\}$: These elements, if chosen, cannot replace the existing i elements in the reservoir. Their contribution is $\frac{k-m}{k} + \frac{m}{k} \times \frac{m-i}{m} = \frac{k-i}{k}$.
- Elements $\forall i \in \{j+1, ..., m\}, a_i$: These elements need to be chosen to be contained in the final sample, and cannot replace the previous $(i-1)$ elements in the reservoir. Their contribution is $\frac{m}{a_i} \times \frac{m-i+1}{m} = \frac{m-i+1}{a_i}$.

- Elements $k \in \{a_m+1, ..., N\}$: These elements cannot be chosen. Their contribution is $\frac{k-m}{m}$.

Multiplying all the contributions terms:

$$P = \prod_{k=m+1}^{N} p_k =$$

$$(\frac{m+1-j}{m+1})(\frac{m+2-j}{m+2})...(\frac{a_{j+1}-2j}{a_{j+1}-j})$$

$$...(\frac{a_{j+1}-j-1}{a_{j+1}-1})(\frac{m-j}{a_{j+1}})(\frac{a_{j+1}-j}{a_{j+1}+1})...(\frac{N-m}{N})$$

$$= \frac{(m+1-j)...(m).(m-j)(m-j-1)...(1)}{(N-m+1)(N-m+2)...N}$$

$$= \frac{m!(N-m)!}{N!} = \frac{1}{C_m^N}$$

We consider the contribution terms in the numerator and denominator separately.

- Denominator: In the denominator, we have terms $\{m+1, m+2, ..., N\}$ (total $(N-m)$ terms) in strictly increasing order.
- Numerator: In the numerator, we have two sets of terms.
  - From elements $\forall i > j, a_i$ in the final reservoir, we have $\{m-j, m-j-1, ..., 1\}$ (total $(m-j)$ terms).
  - From elements $\forall\, (m+1) < k \leq N, k \notin \{a_i\}$, we have $\{m+1-j, m+2-j, ..., N-m\}$ (total $(N-2m+j)$ elements) in strictly increasing order

The terms $\{m+1-j, m+2-j, ..., m\}$ in the numerator exist because they are less than the smallest denominator value $(m+1)$. The terms $\{m-j, m-j-1, ..., 1\}$ come from elements elements $\forall i > j, a_i$. In the denominator, we have terms remaining after the maximum numerator value, $(N-m)$. □

Theorem 3. *Samples of samples retain the property that each element is still selected with equal probability.*

Proof. Say the total data has N elements. First level of reservoir sampling has k elements, and each element in the data has $\frac{k}{N}$ probability of being in the first reservoir sample.

If we want to build a smaller reservoir sample s, such that $s < k < N$ we can in turn perform reservoir sampling on the reservoir sample of size k. Each element of the k-sample will have probability of $\frac{s}{k}$ of being in the smaller sample, and each element of the data set will have $\frac{k}{N} \times \frac{s}{k} = \frac{s}{N}$ probability of being in the smaller sample (perfect random sampling). □

3 Design

In a nutshell, we perform reservoir sampling across the memory hierarchy to induce sequential accesses on persistent storage devices at lower levels. In this section, we describe our design, as well as the assumptions we make.

3.1 Implementation

Figure 6 shows an overview of our architecture. We utilize a hierarchical design over memory which allows addition of multiple data tiers as appropriate for the system in use. On each tier, we are able to perform reservoir sampling repeatedly, while retaining true randomness guarantees at the same time. Apart from the main process which performs the computation required by the Machine Learning algorithm, we have two types of processes running asynchronously that perform data loading in the background:

- **Sample Creators**: Each sample creator is a process that creates a reservoir sample of data points according to the reservoir sampling algorithm. Sampling is performed on just the keys(ordered integers from 1 to the number of images in dataset) first, followed by reading all the necessary data in one go which allows us to optimize for sequential access. We have a single sample creator running at each boundary of the hierarchy.
- **Batch Creators**: A single batch creator process consumes data points from reservoir samples in main memory, and generates batches ready to load by the ML program.

We have implemented *OneAccess* as a Python library [10] compatible for use with PyTorch [1]. The implementation is structured in two main parts: data storing and data loading. The former involves storing the dataset on persistent storage and generating an intermediate representation (IR) with a few large files holding the data. Data loading involves the sample creator and batch creator processes working together to generate batches for the machine learning algorithm.

We would like to describe the following important decisions made while designing *OneAccess*:

- **Intermediate representation (IR)**: We generate an IR for each dataset, where all data is stored in a few large files of the order of GBs, where each file consists of multiple large numpy arrays. The purpose of generating an IR is to ensure sequential accesses in the storage media. In contrast, storing image data as separate files for instance, will lead to random accesses during reservoir sampling which would be undesirable.

[1] The implementation for OneAccess is available online on GitHub here: `https://github.com/aarati-K/one_access`

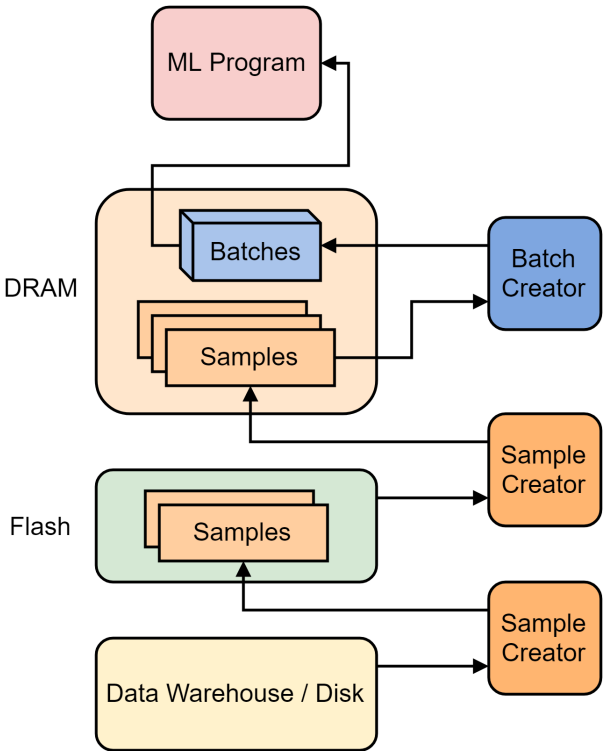

Figure 6: Architecture of OneAccess

- **Concurrent data structures**: We use *Queues* [8] for communication between all concurrent processes. We have taken precautions to avoid lock contention, by using multiple queues for holding samples.
- **Eager sample creation**: We provide support for creating sample(s) in advance before starting training epochs. This means that the computation will not have to wait on batch creation when the training begins.
- **Parameterization for flexibility**: We have parameterized various aspects of the implementation like sample size, batch size, number of samples, etc for ease of experimentation and configuration.
- **Support for sampling without replacement**: Reservoir sampling is traditionally performed with replacement. We have also incorporated the option of sampling without replacement (which is the default configuration).

3.2 Assumptions

In designing OneAccess, we have been led by assumptions that offer both challenges and opportunities.

- Creating an intermediate representation is one of our design decisions to ensure the sequential access. To obtain better performance, each IR file should be of the order of GBs. The IR generation is different for different datasets, and we assume that the implementation is guided by by the goal of generating a few large files.
- In the IR, we assume that all the data points are of the same size. This does not necessarily hold true for image datasets (like MS-COCO), and we apply transfroms (such as cropping) to generate a constant value size.
- We assume that users have good knowledge of choosing the proper sample size with respect to the batch size. We suggest users to choose a sample size in terms of multiples of batch size and the value in order of 100X-1000X.
- Our design is based on the assumption that main memory is best suited for random accesses over all the components in the storage hierarchy. All random accesses are performed in main memory.

3.3 API and Programming

```
import OneAccess
import MachineLearningModel
import math

data_store =
    OneAccess.store.Cifar10(
    input_data_folder=".....",
    max_batches=1,
    batch_size=1,
    rel_sample_size=10,
    max_samples=1,
    transform=[])

# Setup up IR, metadata,
# and initial sample
data_store.initialize()
data_loader =
    OneAccess.load.DataLoader(
        data_store, epochs=1)

model = MachineLearningModel()
for i in range(data_loader.num_batches):
    batch = data_loader.get_next_batch()
    model.train(batch)
```

4 Evaluation

We have benchmarked *OneAccess* on two datasets, being Cifar-10 [3] and MS-COCO [7], and compared the performance against PyTorch. We have also evaluated our

system for end-to-end training with Hogwild! [9] on the Cifar-10 dataset. We discuss the results obtained and our observations in this section. We have considered a two level memory hierarchy, with a main memory (of size 32GB) and an SSD (Samsung 960 EVO, 500GB).

4.1 Cifar-10

The total size of the Cifar-10 dataset is 163MB, with 50,000 images of equal size. We generated a single IR file of size 163MB containing all the images, and the total time to create the IR was 1.5s.

The first benchmark experiment we run is on the reservoir sample creation, to understand the tradeoffs with choosing different sample sizes. Figure 7 shows the time taken to create samples of increasing size, for ten different instances of sample creation. For a sample of size 1, we observe a wide variation in the sample creation time, whereas the spread becomes lower and the sample creation time becomes roughly constant for sample sizes 10 and above.

Through this experiment, we conclude that a major component of sample creation time is reading the IR file sequentially. Two observations reinforce this claim. First, the wide variation in sample creation time for sample size 1 is because different amount (numpy arrays) of data will be read from the IR file to access the sampled point. Second, the sample creation time changes very little for sample sizes 10 and up, indicating that the whole file needs to be read to generate the sample. Thus, as a rule of thumb, we propose that *it is better to generate as large a reservoir sample as possible*.

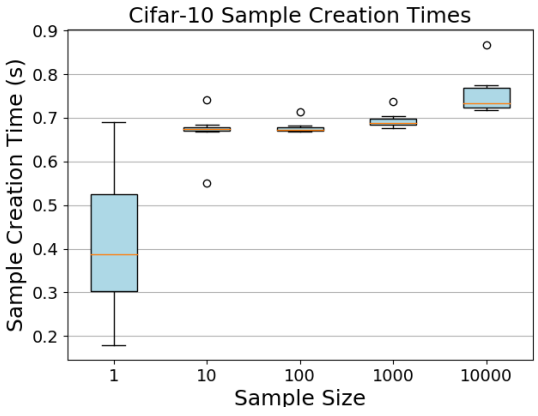

Figure 7: Cifar10 Sample Size by Creation Time: we are able to achieve almost constant sample size.

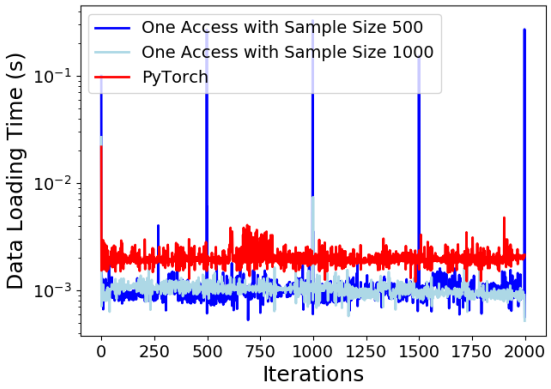

Figure 8: Cifar10 Iterations by Data Load Time

Framework	**Total Time (s)**
Pytorch	4.02
OneAccess (Relative sample size 500)	4.01
OneAccess (Relative sample size 1000)	2.72

Table 5: Cifar-10: Total time taken to fetch batches for one epoch (batch size 4). OneAccess is 31.4% faster for relative sample size 1000 (sample size 4×1000).

Next, we compare the time taken to fetch batches of size 4 through OneAccess vs PyTorch[2] for one epoch. The results of this measurement are shown in Figure 8. For a relative sample size of 500 (sample size 4×500), we observe peaks at intervals of 500, indicating that batch creation blocks on sample creation on these iterations. Similarly, for a relative sample size of 1000, we observe peaks at intervals of 1000. This observation further reinforces our proposition to use samples as large as possible. Table 4.1 indicates the total time taken to fetch batches for one epoch. OneAccess with relative sample size of 1000 is 31.4% faster.

4.2 Multiple Client model on Cifar-10 using HogWild!

In Hogwild, multiple worker processes are allowed to update a common shared model without locking, with the possibility of workers overwriting each other's updates. Typically, each worker itself creates another worker for data loading, which involves the additional overhead of utilizing double the processes. Unlike this general implementation, we implement Hogwild with OneAccess by creating a single batch (and sample) creator process.

[2]We use a different implementation from the Cifar-10 dataset module distributed by PyTorch. In the implementation used, each Cifar-10 train image is contained in its own file instead of loading the whole dataset into the process memory.

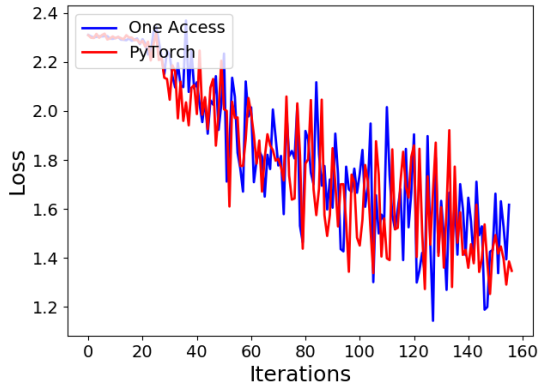

Figure 9: Convergence for Hogwild on OneAccess vs. PyTorch

Framework	Total Time (s)
Pytorch	127
OneAccess (Relative sample size 100)	83.85
OneAccess (Relative sample size 500)	53.71
OneAccess (Relative sample size 1000)	50.5

Table 6: Statistics for end to end training time on Cifar-10 using HogWild!

The common batch queue is shared with all the workers, thus reducing the overhead.

Table 4.2 shows the total training time for one epoch of OneAccess against the vannila implementation in PyTorch with each worker having its own data loader. The training is performed with two workers and a batch size of 32 for one epoch. We see that OneAccess is 2.5x faster than PyTorch in the best case. Figure 9 shows the error rate of a worker with OneAccess and PyTorch. The reduction in error rate is about the same, which shows that OneAccess is able to provide the same amount of randomness.

4.3 MS-COCO Detection

Similar to Cifar-10, we have benchmarked **OneAccess** for the MS-COCO dataset. This dataset is 18GB in size, with close to 118K images of variable sizes. Thus, to generate an IR for MS-COCO, we had to pre-process the images to crop them to size 224×224. We generated nine IR files, each of size roughly 2GB. The total time taken to create the IR for MS-COCO was about 23 minutes.

Figure 10 shows the time taken to create samples of increasing size. Similar to Cifar-10, we observe very little increase in sample creation time for sample sizes 100 and up. We compare the performance of OneAccess against the MS-COCO dataset module provided by PyTorch. Figure 11 shows the time taken per iteration, while fetching batches of size 32 for one epoch. For OneAccess, we observe that batch creation blocks on sample creation periodically. In contrast, PyTorch (with one worker) takes almost constant time to fetch batches. Table 4.3 indicates the total time taken, and OneAccess is 3.6x faster than PyTorch in the best case.

Framework	Total Time (min)
Pytorch (with 1 worker)	23
PyTorch (with 2 workers)	12
PyTorch (with 4 workers)	6.8
OneAccess (Rel. sample size 400)	6.4

Table 7: MS-COCO: Time taken to fetch batches of size 32 for one epoch. OneAccess is 3.6x and 1.9x faster compared to PyTorch with 1 and 2 workers respectively.

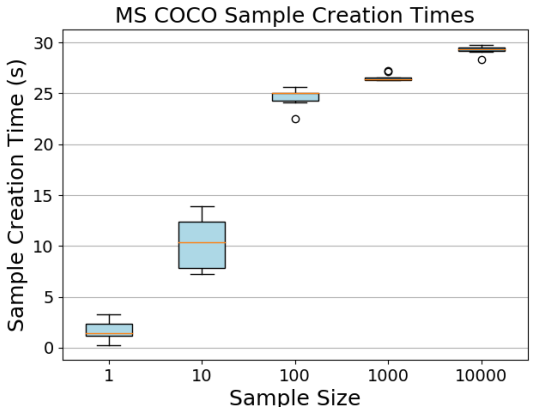

Figure 10: COCO Sample Size by Creation Time: we are able to achieve almost constant sample size.

5 Related Work

Work by Tu et al. [11] discusses a number of theoretical results on how access patterns affects convergence rate for machine learning algorithms. Primarily, the authors observe by studying fixed-partition and random-sampling that sampling strategy can drastically affect the convergence rate. They find that random-sampling is vastly superior to fixed-partition sampling.

For handling the distributed setting, we refer to two algorithms from [2], namely *CodedShuffle* and *UberShuffle*. The idea of coded shuffling is that instead of transmitting data points one by one, the master node linearly combines multiple data points, and broadcasts a fewer number of *coded* data points. *UberShuffle* improves on the coded shuffling algorithm by having the nodes in the distributed system network encode within themselves, and pass on packets to each other more efficiently.

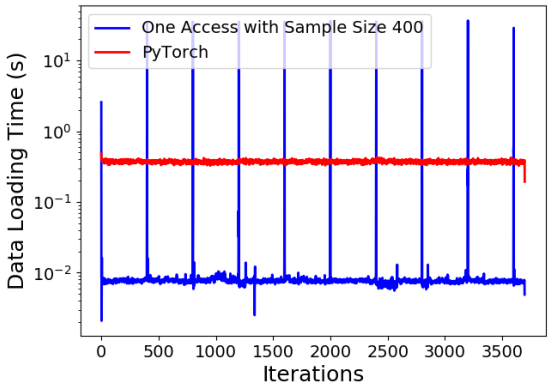

Figure 11: MS COCO Iterations by Data Load Time

We have seen the usage of *Reservoir sampling* in various ML scenarios. Work by Xixuan et al. discussed a unified architecture for in-RDBMS analytics. The two key factors that they found impact performance are the order data is stored, and parallelization of computations on a single-node multicore RDBMS. In some cases, a single shuffle of the data may be too expensive (e.g., for data sets that do not fit in available memory). To cope with such large data sets, they have used *Reservoir Sampling* to create subsamples.

Similarly, *Weka* is a collection of machine learning algorithms for data mining tasks. In *Weka*, Reservoir sampling is being used in non-incremental learning algorithms on large datasets. This helps to overcome limited DRAM and Java heap space availability.

Also, we would like to mention other data loaders from few popular ML frameworks. *PyTorch* provides a base dataloader with some helper functions to load data, shuffling, and augmentations. This comes with an option to set number of workers to load the data in parallel or to fasten the data loading process. But this doesn't solve the problem of random access and also there is no efficient way to support multiple clients using the same dataloader.

TensorFlow is another framework that has a similar API to *PyTorch* dataloader called *tf.data*. This API enables users to build complex input pipelines from simple, reusable pieces. We did not compare our model with *TensorFlow's* data API, but we believe ours would outperform *TensorFlow's* dataloader as we have sequential access and also allow users to apply transformations in parallel. We would love to compare with *TensorFlow* and will work on this in the future.

6 Future Work

Through our first implementation of OneAccess, we observe substantial gains for data loading (as high as 3.6x) over PyTorch. Our framework uses reservoir sampling, and exploits sequential accesses while at the same time retaining perfect randomness guarantees.

Our evaluation in this work was limited to a two-level memory hierarchy consisting on main memory and an SSD. It would be interesting to extend our experiments to a multi-level hierarchy with larger datasets. One limitation of our current implementation is the assumption of a constant value size while generating the intermediate representation. This does not necessarily hold true and can possibly require additional (possibly irreversible) pre-processing of data. It might be good to look at other representations which do not have this constraint.

Another possible direction of work would be to automatically determine a good choice of sample size for a given task. This decision is influenced by many factors, like the training algorithm being used, the compute hardware, the components in the storage hierarchy, and the memory available. It might be possible for the system to benchmark the platform by running a few iterations, and then decide on these parameters on the fly.

OneAccess is simply the first version in an extensible system, and is currently implemented for a single machine. We envision that the final version will be distributed for large-scale application, and will have dataset versioning support natively implemented, thus improving the synergy between machine learning training and loading systems.

References

[1] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *CoRR* abs/1704.04760 (2017). arXiv: 1704.04760. URL: http://arxiv.org/abs/1704.04760.

[2] Jichan Chung et al. "UberShuffle: Communication-efficient Data Shuffling for SGD via Coding Theory". In: ().

[3] *Cifar-10 dataset*. URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[4] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[5] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. Citeseer, 2009.

[6] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[7] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312.

[8] *Multiprocessing Best Practices*. URL: https://pytorch.org/docs/stable/notes/multiprocessing.html.

[9] Feng Niu et al. "HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent". In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. NIPS'11. Granada, Spain: Curran Associates Inc., 2011, pp. 693–701. ISBN: 978-1-61839-599-3. URL: http://dl.acm.org/citation.cfm?id=2986459.2986537.

[10] *OneAccess*. URL: https://github.com/aarati-K/one_access.

[11] S. Tu et al. "Breaking Locality Accelerates Block Gauss-Seidel". In: *ArXiv e-prints* (Jan. 2017). arXiv: 1701.03863 [math.OC].