Is Systematic Data Sharding able to Stabilize Asynchronous Parameter Server Training?

Nikodimos Provatas
Computing Systems Laboratory
National Technical University of Athens
Athens, Greece
nprov@cslab.ece.ntua.gr

Ioannis Konstantinou
Dept. of Computer Science
and Telecommunications
University of Thessaly
Lamia, Greece
ikons@uth.gr

Nectarios Koziris
Computing Systems Laboratory
National Technical University of Athens
Athens, Greece
nkoziris@cslab.ece.ntua.gr

***Abstract*—Over the last years, deep learning has gained an increase in popularity in various domains introducing complex models to handle the data explosion. However, while such model architectures can support the enormous amount of data, a single computing node cannot train the model using the whole data set in a timely fashion. Thus, specialized distributed architectures have been proposed, most of which follow data parallelism schemes, as the widely used parameter server approach. In this setup, each worker contributes to the training process in an asynchronous manner. While asynchronous training does not suffer from synchronization overheads, it introduces the problem of stale gradients which might cause the model to diverge during the training process. In this paper, we examine different data assignment schemes to workers, which facilitate the asynchronous learning approach. Specifically, w e propose two different algorithms to perform the data sharding. Our experimental evaluation indicated that when stratification is taken into account the validation results present up to $6X$ less variance compared to standard sharding creation. When further data exploration for hidden stratification is performed, validation metrics can be slightly optimized. This method also achieves to reduce the variance of training and validation metrics by up to $8X$ and $2X$ respectively.**

***Index Terms*—deep learning, distributed training, data management**

I. Introduction

Over the last years, deep learning has become a field of increasing interest and popularity. Neural networks have widely been adopted in a variety of big data data applications. For instance, neural networks are widely used in the image classification [1, 2], the speech recognition [3, 4] and natural language processing [5, 6] and other fields [7, 8]. In the first two applications, convolutional and recurrent neural network are used respectively for classification purposes.

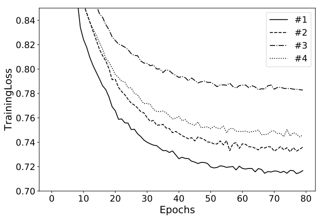
The wide use of neural networks in various domains, as those stated above, followed by the vast amount of data that emerges, has led to the creation of more complex and deep model architectures. This complexity is necessary to create more accurate models for such large datasets. For example, Microsoft has proposed the ResNet architecture in 2015 [9] which achieves to effectively classify the ImageNet Dataset [10]. During the same year, Digital Reasoning has proposed an 160 billion parameter network [11] specialized for natural language processing tasks.

While the constantly increasing amount of data can be used to create more accurate models with complex network architectures, the total size of the data and the neural networks prohibit training on a single machine. Thus, there have been proposed ways to train models in a distributed fashion. Multiple systems are designed to support distributed deep learning, as Google TensorFlow [12], and PyTorch [13] (original developed by Facebook). Moreover, deep learning libraries have also been developed for general purpose distributed systemssd BigDL [14], a framework designed for Apache Spark [15, 16]. Other systems, as Uber's Horovod [17], have been designed to provide ease of use on the distributed training using scripts designed for single-node training.

Distributed deep learning systems follow either the *model* or the *data parallelism* approach to create neural network models. On the one hand, *model parallelism* [18, 19] implies that the model is distributed across different machines, so that each part of the model fits in the memory of each worker. On the other hand, *data parallelism* [20] is used to handle the increasing volume of the data, where data are split into shards assigned to different cluster machines. Each machine is used to train a global model only with the data assigned to them. Note that nowadays the vast majority of distributed deep learning systems support data parallelism, while it can be applied the same regardless of the model used for the training [21].

The aforementioned systems can use the data parallelism scheme under different architectures. In some systems, as PyTorch and TensorFlow, workers can operate synchronously, organised in *all-reduce rings* to combine their training updates [22, 23]. Another common architecture is the *parameter server* one, where a global model is trained either synchronously or asynchronously with gradients computed from different workers. This architecture is supported by TensorFlow [24] for one of the provided programming APIs, while PyTorch provides all the essential building blocks to implement the parameter server case [22].

Due to the lack of synchronization, asynchronous parameter server learning does not suffer from any related time overheads [25]. However, it is sensitive to conflicting and stale updates, which may not favor the quality of the resulting model [26, 27].


(a) Training Loss

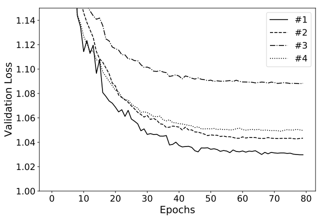

(b) Validation Loss

Fig. 1: Loss of multiple distributed training runs of a simple CNN network over CIFAR-10 dataset.

In this paper, we examine whether systematic data sharding is capable of further enhancing the asynchronous training approach. Up to now, data are usually randomly assigned to workers regardless of the training mode. However, one question that emerges is whether random assignment further affects the quality of the asynchronous learning [28]. This approach is compared with two others presented in section III, which take into account obvious and hidden stratification patterns.

As a motivation experiment, we train a simple 4-layer CNN network over the CIFAR-10 dataset multiple times, having randomly distributed the data to the workers before each run. Figure 1 presents the training and the validation loss per experiment. In this figure, we notice the existence of variance between the loss values from the subsequent runs. This observation further motivated our research over whether various sharding techniques lead to more stable results. Note that when machine learning is applied in domains, as health applications, it is important that the resulting model presents small variance in the classification accuracy. For instance, in the case of predicting the risk of diabetes, reliability model metrics depend on the standard deviation, and therefore the variance, of the model's accuracy [29]. Thus, to make more reliable models under asynchronous learning, it is crucial to further reduce the variance in the resulting metrics.

The main contributions of this paper are the following:

- We propose two systematic sharding approaches, the Stratified and the Distribution Aware approach.
- We provide a detailed experimental evaluation as a proof of concept of how and why systematic data sharding can stabilize the asynchronous learning process. When stratification is considered, variance of validation metrics can be reduced by up to $6X$. Distribution Aware sharding can enhance training and validation metrics in most cases with up to $8X$ and $2X$ less variance respectively.

The rest of the paper is organized as follows: Section II presents any preliminary information necessary for the reader to fully understand the paper. Section III describes the proposed algorithms, followed by a detailed experimental evaluation in IV. Finally, in V and VI, we present related work and summarize our contributions.

II. Preliminaries

In this section, we will discuss any preliminary knowledge necessary for the reader to fully understand this paper. Table I refers to all the related terms and symbols that are used throughout this paper.

TABLE I: Terms and symbols used in the paper

Term/ Symbol	Description
Server	A parameter server task of the parameter server architecture
Worker	A worker task of the parameter server architecture
Shard	Subset of the dataset assigned to a worker
Mini-Batch Size (B)	The number of training examples used in one iteration of SGD (globally)
Per-Worker Mini-Batch Size (WB)	The number of training examples a worker uses from the local shard to apply the SGD per each iteration
Learning Rate (α)	Initial learning rate for single-worker case
Learning Rate (α_w)	Learning rate per worker for the distributed case
Epoch	A full training pass on the dataset
Global Model	Model stored in servers
Evaluator	Task evaluating the global model

A. Deep Neural Networks and Stochastic Gradient Descent

Deep neural networks [30] aim to approximate some function $f : \mathbb{R}^n \to \mathbb{R}$. In the context of classification problems, this function is used to classify a feature vector $\vec{x}$ to a category y, i.e. $y = f(\vec{x}; \vec{w})$, where $\vec{w}$ is used to denote the neural network parameters. Neural networks are organized as a sequence of layers, which are sequentially connected. Thus, the output of one layer becomes the input of the following one. Depending on the task, different layer types have been proposed, as convolutional layers for image classification [1].

Using neural networks for classification, requires that the neural network is trained to provide the best approximation of the aforementioned function f in respect to its parameters $\vec{w}$. The set of model parameters $\vec{w}$ that can provide such approximation is the minimizer of a *loss function*, which describes the error of a model given a set of example feature vectors $\vec{x}_1, \vec{x}_2, ..., \vec{x}_n$ and the corresponding predictions $y_1, y_2, ..., y_n$. This set of parameters $\vec{w}$ is located by the Stochastic Gradient Descent (SGD) algorithm or any of its variants [31, 32], probably accelerated by the momentum method [33]

Let $L : \mathbb{R}^n \to \mathbb{R}$ be the loss function used on training a model with parameters $\vec{w}$. The k-th iteration of SGD [34], based on a randomly selected training example $(\vec{x}_k, y_k)$, is mathematically formulated as follows:

$$\vec{w}_{k+1} = \vec{w}_k - \alpha \cdot \nabla_{\vec{w}_k} L(\vec{w}_k; \vec{x}_k, y_k) \quad (1)$$

where w_i and a stand for the model parameters in the i-th iteration of the SGD and the learning rate respectively.

SGD is more commonly used in the form of mini-batch SGD. In mini-batch SGD, a random set of n training examples $(\vec{x}_1, y_1), (\vec{x}_2, y_2), ..., (\vec{x}_n, y_n)$ is used to update the model parameters in each training iteration. The update is performed based on the average of the gradients computed by each training example and equation 1 is rewritten as follows:

$$\vec{w}_{k+1} = \vec{w}_k - \frac{\alpha}{n} \cdot \sum_{i=1}^{n} \nabla_{\vec{w}_k} L(\vec{w}_k; \vec{x}_i, y_i) \quad (2)$$

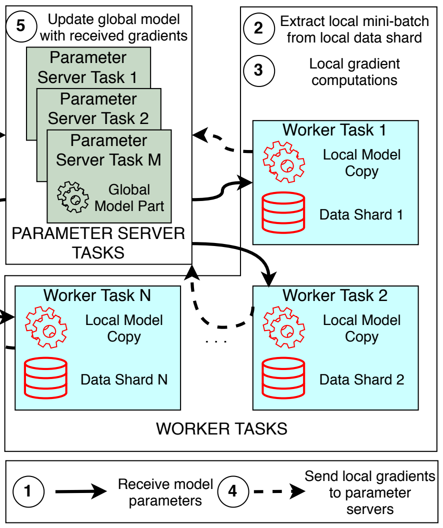


Fig. 2: Parameter Server Architecture

The number of training examples that the mini-batch consists of is called *mini-batch size*.

B. Parameter Server Architecture

Training deep neural networks, and machine learning models in general, can be scaled if multiple devices are available. A state-of-the-art architecture designed for this purpose is the *parameter server* one [19, 35, 36], used from a variety of ML-related systems, as TensorFlow [12]. To train a deep neural network under the parameter server, a set of *parameter server* and *worker* tasks are deployed. Parameter server tasks are responsible for storing and updating the global model [37]. On the contrary, worker tasks use the SGD optimization algorithm to compute gradient updates on their local copies of the model, used by the parameter server tasks to update the global one.

The whole training procedure, when the parameter server approach is used, is fully depicted in Figure 2. At first, a worker task updates its local model parameters from the latest global ones received from the parameter server tasks (Step 1). Step 2 includes a worker extracting a mini-batch of data points from the subset of data that was assigned to them, namely a *data shard*. Note that the data shards of each worker are disjoint. Sequentially, the extracted mini-batch is used to compute a set of gradients of the loss function on their local copy of the model (Step 3). Having computed these gradients, they are sent back to parameter servers (Step 4). Finally, in Step 5, the parameter servers update the global model with the received gradients. The aforementioned procedure is repeated until the model has been trained for a user-defined number of epochs or until other criteria are met (e.g. early stopping [38]).

C. Hyperparameters in Distributed Training

Before training a deep learning model, it is necessary to identify the set of hyperparameters for both the layers of the neural network and the various parameters that affect the training, as the mini-batch size B, the learning rate a, etc. *Hyperparameter tuning* is very challenging, since data affect the right choice of hyperparameters [39].

Even if the hyperparameters have been carefully selected for a single-node training of a neural network, they cannot be taken as granted for the case of the distributed training. In case the distributed training follows the parameter server architecture, instead of attempting to tune the hyperparameters for each distributed training setup, crucial hyperparameters, as the per-worker mini-batch size WB and the learning rate α, can be computed from the hyperparameters used when performing single-node training [40].

Suppose that the best hyperparameters to train a neural network on a single-node setup are B for the mini-batch size and α for the learning rate. Let the train be performed on a cluster following the parameter server architecture. Regarding the WB hyperparameter, its product with the number of workers has to be constant. Given that single-node training has one worker and B is the best mini-batch size, in a distributed setup with n workers, WB can be adapted as follows:

$$B = n \cdot WB \iff WB = \frac{B}{n} \tag{3}$$

Furthermore, the learning rate α_w of a distributed setup can be computed from the learning rate α of the equivalent single-node training following equation 4.

$$\alpha_w = \frac{\alpha}{n} \tag{4}$$

D. Asynchronous Training

The parameter server architecture, presented in subsection II-B, is widely used with workers updating the global model in an asynchronous manner (see introductory section I). In asynchronous learning each worker computes gradients using the global parameters parameter that they received from the server upon their last gradient updates. It is important to mention that, on the same training step, the workers may well use different parameters in order to compute the new gradients. Under this case, equation 2 is rewritten as shown in equation 5, where $\vec{w}_\theta$ denotes the parameters retrieved from the servers to compute the gradients and $\vec{w}_k, \vec{w}_{k+1}$ the current and the resulting global parameters respectively.

$$\vec{w}_{k+1} = \vec{w}_k - \frac{\alpha_w}{WB} \cdot \sum_{i=1}^{WB} \nabla_{\vec{w}_\theta} L(\vec{w}_\theta; \vec{x}_i, y_i) \tag{5}$$

Note that workers using older parameters causes stale gradients, mentioned in the introductory section I, which might cause the model to diverge. The learning rate adaption, shown in equation 4 is performed to handle the staleness phenomena, by reducing each worker's contribution on the global model.

III. Approaches

In distributed learning, under the data parallelism approach, workers contribute to the global model with gradients computed using different subsets of the data, as it was decribed in Section II. Currently, distributed deep learning systems randomly assign data to workers. TensorFlow, for instance, uses the *shard* mechanism in its data pipeline [41], which assigns training examples to workers in a round-robin fashion.

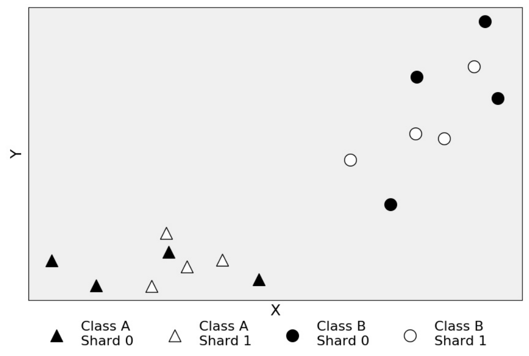


Fig. 3: Stratified assignment: Half of the points from each class are assigned to each shard.

Since the purpose of this paper is to compare the effect of data assignment techniques to workers, we create a custom mechanism that creates the shards offline according to the method the user prefers. Note that it is safe to create the shards beforehand, since the equivalent mechanism of TensorFlow described, suggests to be applied at first. Thus, the data assignment is static upon the training process.

In this paper, we propose an offline data assignment system. This system takes as input the data set, located in a shared or distributed file system, from which we want to create the various shards, the number of workers (N) that participate in the training process and the algorithm that will assign training examples to workers followed by any related parameters. Having provided this information, the related algorithm is used to create N shards from the dataset, by assigning worker indicative labels to each training example.

In this paper, we examine three different algorithms regarding the data shard creation process. As a baseline, a random data assignment to workers is used.

A. Stratified Sharding

Random data assignment cannot ensure that each worker will have an equivalent view on the train data set. For instance, let us present a scenario where only two classes are present in the train data set, where each class represents half of the available data. Depending on the order of the training examples, some workers may well have available more examples from the one class compared to those from the other. Thus, the workers will not have a representative view of the world that these data describe, but each one will recognize one class as dominant regarding its frequency. In the aforementioned scenario, workers may attempt to lead the parameters of the global model to a different direction from each other, such that the model will more efficiently categorize the examples available in their local shard, affecting the efficiency of the resulting model.

Inspired from machine learning basics, we propose stratified assignment as another technique, which is able to guarantee that each worker will be provided with a data shard that is equivalent to the world the whole train set describes. The aforementioned equivalence refers to each shard consisting of the same percentage from each class compared to the one on the whole train data set. The result of creating shards using the stratified approach is illustrated in Figure 3.

```
1:  procedure StratifiedSharding($x, y, n$)
2:                    ▷ x is an array of multidimensional tensor representing training examples
3:                    ▷ y is the correspongin array of labels
4:                    ▷ workers have identifiers 0, 1, ..., n-1
5:      $classes = unique(y)$                ▷ Identify available classes
6:      for each class in classes do
7:          $x\_class, y\_class = in\_class(x, y, class)$
8:          $class\_size = length(y\_class)$
9:          for $i = 0$ to $class\_size - 1$ do
10:                                ▷ Round Robin split for each class
11:             $worker\_id = i \bmod n$
12:             Assign example $x\_class(i)$ to $worker\_id$
13:             Assign label $y\_class(i)$ to $worker\_id$
14:         end for
15:     end for
16: end procedure
```

Fig. 4: Stratified data sharding to workers using mod

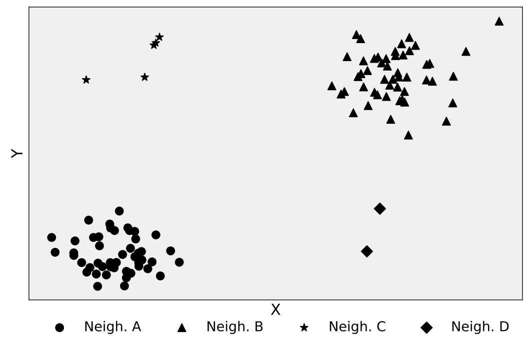


Fig. 5: Example of 2-dimensional points organized in dense (A, B) and sparse (C, D) neighbourhoods.

Figure 4 outlines the algorithm used to perform a stratified assignment. The first step is to find the set of distinct classes, given the array of labels (line 5). Sequentially, we retrieve the set of training examples in each class (line 7). Finally, for every such set the data are given in a round robin manner to the workers (lines 9 - 13). This approach guarantees that each class percentage remains intact in the sample compared to the one on the whole train set.

B. Distribution Aware Sharding

Stratified assignment is an obvious way to provide the participating workers with an equivalent view of the data. However, apart from the obvious class stratification, hidden stratification may also exist in the data set. Data can be further categorized to neighbourhoods (clusters), based on their position in respect to other data points.

Distribution aware sharding takes into account the neighbourhoods in the same manner the stratified one uses the classes. Figure 5 illustrates an example of neighbourhoods in the 2-D space, where some (A, B) consist of adequate points and others (C, D) of isolated examples. It is important to mention that we cannot determine whether this isolated points are outliers or if the available data set does not consist of more equivalent points. Therefore, outlier points forming sparse neighbourhoods should be handled in a different way. Thus, our algorithm distinguishes two cases:

```
procedure DistrAwareSharding($x, y, n, classes$)
                                        ▷ x is an array of multidimensional tensor representing training examples
                                        ▷ y is the correspongin array of labels
                                        ▷ workers have identifiers 0, 1, ..., n-1
                                        ▷ classes is the number of classes for the KMeans algorithm
    $x\_flattened = flatten(x)$         ▷ Flatten each example
    $x\_distribution = PCA(x\_flattened)$
    $cluster\_ids = KMeans(x\_distribution, classes)$
    $clusters = unique(cluster\_ids)$   ▷ Identify clusters
    for each cluster in clusters do
        $x\_cluster, y\_cluster = in\_cluster(x, y, cluster)$
        $cluster\_size = length(y\_cluster)$
        if $cluster\_size > n$ then
            for $i = 0$ to $cluster\_size - 1$ do
                                        ▷ Round Robin split for each cluster
                $worker\_id = i$ mod $n$
                Assign example $x\_cluster(i)$ to $worker\_id$
                Assign label $y\_cluster(i)$ to $worker\_id$
            end for
        else
            Assign each x(i) in $x\_cluster$ to all workers
            Assign each y(i) in $y\_cluster$ to all workers
        end if
    end for
end procedure
```

Fig. 6: Distribution aware data sharding to workers using mod

1) *Densely populated neighbourhoods:* Data points from such neighbourhood shall be assigned in a round robin fashion to workers. This assignment provides each worker an equivalent view of this neighbourhood.
2) *Sparsely populated neighbourhoods:* Each worker should be also provided with a view from such neighbourhoods. Since the data points are insufficient to split, we broadcast the whole neighbourhood to all workers.

Distribution aware assignment is fully outlined in Figure 6. First, in order to have data points in $\mathbb{R}^n$, we flatten each training example (line 6). KMeans algorithm [42, 43] is used to identify the various neighbourhoods that can be formed by the available data (line 8), using as input a summarized version of the data set with minimal loss, which resulted by applying PCA) [44] (line 7). Finally, a round robin assignment is performed for each densely populated neighbourhood (lines 11-19), while sparsely populated neighbourhoods are broadcast to all workers (lines 21-22). Note that we consider sparse neighbourhoods those with less points than the number of workers. This assumption is made, since otherwise each worker cannot have an equivalent view on these neighbourhoods.

C. Time Complexity

Having presented our sharding approaches, it is important to identify their time complexity. Suppose a dataset of n training examples in the $d-$dimensional space is available. The baseline random assignment can be implemented in $\mathcal{O}(n)$, by choosing to assign each example to one of the workers with the less assigned data. Stratified sharding is implemented with two nested for loops, passing the data points once (see Figure 4). Thus, stratified sharding also is a $\mathcal{O}(n)$ algorithm. Regarding the distribution aware approach, its computational complexity is determined by applying the PCA and KMeans algorithms, which are used from the scikit-learn library in our implementation. PCA has a $\mathcal{O}(n \cdot d^2)$ [45] time complexity. KMeans, since we fix a number of maximum iterations T to 150, becomes a $\mathcal{O}(k \cdot n)$ [46] algorithm, where k is the desired number of neighbourhoods. Therefore, the distribution aware sharding is a $\mathcal{O}(n \cdot max\{d^2, k\})$ algorithm.

IV. Experiments

In this section, we provide an extensive experimental evaluation of the algorithms presented in section III.

A. Experimental Setup

We conduct our experimental evaluation with a cluster consisting of 15 virtual machines. Each virtual machine has 4 CPUs and 16GB RAM and operates with Ubuntu 16.04.6 LTS. We use TensorFlow to build and train the models under the parameter server architecture. As a common distributed file system, we use Apache Hadoop [47] deployed with 1 namenode and 14 datanodes.

Regarding the tasks that participate in the training, each one is deployed in a different VM. In further detail, we deploy 2 servers, 12 workers and 1 evaluator task. In the training process, each worker exploits only data assigned to them from the sharding algorithm.

Note that there are no GPUs available on our experimental cluster. However, since we want to evaluate how the model quality under an asynchronous training setup is affected from the algorithm used to create the data shard, the lack of such accelators is not crucial for the quality of our experiments.

B. Datasets, Networks and Training Setup

We evaluate the proposed data sharding schemes with benchmarks from the Image Classification domain. The experimental evaluation for the different data assignment strategies is performed on training the ResNet-56v1 network [9] with the CIFAR-10 and CIFAR-100 [48] respectively. CIFAR-100 is examined both with the coarse and fine grain labels available. Regarding the network configuration, [9] clearly proposed a set of hyperparameters for training on the CIFAR datasets in the single node case. Taking the aforementioned single node training configuration into account, we adjust the necessary hyperparameters for the distributed setup (see Section II-C).

For each experiment we perform five runs and provide the mean values and the variance of the final loss and accuracy over the training and the validation set. We further discuss any effects that occur by using each sharding approach. Regarding the Distribution Aware technique, we consider the same global data set size as the baseline. In this way, we do not further train the model with more mini - batches per epoch, but study how sparse neighbourhoods can affect the training results.

C. CIFAR10

1) ***Metrics and Variance.:*** Table II outlines the mean and variance of training and validations metrics for each sharding approach applied on the CIFAR-10 dataset. While using stratified sharding led to similar reduction of the train loss, it is important to notice its effect on the variance of the

TABLE II: Statistics (5 runs) of final Training and Validation Loss / Accuracy on the CIFAR-10 dataset per method

Method			**Training Metrics**				**Validation Metrics**			
			Loss		Accuracy		Loss		Accuracy	
			Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
Random			0.165786	0.001567	0.920000	0.001086	0.442370	0.006174	0.935489	0.000863
Stratified			0.165751	0.001312	**0.922397**	0.000736	0.444076	0.005401	0.935396	**0.000284**
Distribution Aware	Class Number	20	0.165237	0.002471	0.921583	0.004059	0.445127	0.004059	0.935291	0.000648
		30	0.165166	0.001270	0.922218	**0.000128**	0.442075	**0.003431**	**0.935885**	0.000550
		40	**0.165046**	**0.000761**	0.921884	0.000851	**0.441692**	0.011402	0.935127	0.001358

TABLE III: Sparsely populated neighbourhoods through Distribution Aware Algorithm (various cluster sizes as input) with the resulting validation metrics for 3 of the runs (CIFAR-10). Mean Size refer to the mean population of all those neighbourhoods.

#Clusters	**Run #1**			**Run #2**			**Run #3**		
	Sparse Neighb.	Mean Size	Valid. Accuracy	Sparse Neighb.	Mean Size	Valid. Accuracy	Sparse Neighb.	Mean Size	Valid. Accuracy
20	2	1	0.9352	1	1	0.9345	3	1	0.9361
30	6	2.83	0.9351	6	1.17	0.9361	4	1.25	0.9364
40	7	1.57	0.9370	8	1	0.9345	11	1.56	0.9330

TABLE IV: Statistics (5 runs) of final Training and Validation Loss / Accuracy on the Coarse CIFAR-100 dataset per method

Method			**Training Metrics**				**Validation Metrics**			
			Loss		Accuracy		Loss		Accuracy	
			Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
Random			0.352169	0.004761	0.811742	0.003103	1.216487	0.010065	0.805993	0.004452
Stratified			**0.346032**	0.001903	**0.814008**	0.000766	1.232794	**0.006096**	0.805841	**0.000728**
Distribution Aware	Class Number	30	0.349379	0.003350	0.812987	0.002031	1.219970	0.009967	0.806072	0.000931
		40	0.350087	**0.001738**	0.812330	**0.000535**	1.211612	0.008745	**0.806237**	0.002374
		60	0.347240	0.001893	0.812624	0.001138	**1.209313**	0.021793	0.805775	0.003009

metrics. Stratified sharding concludes in training and validation accuracy with $1.47X$ and $3.03X$ less variance compared to the baseline. Loss values also follow similar patterns.

Distribution aware mechanism leads to a slight enhancement in the metrics and might further the decrease of the variance. For instance, when sharding using 30 neighbourhoods, the validation accuracy meets the best value of 0.9359. Training accuracy is enhanced by 0.22% with a $8.48X$ less variance compared to random. Compared to the baseline, variance of validation metrics is also enhanced by up to $1.57X$.

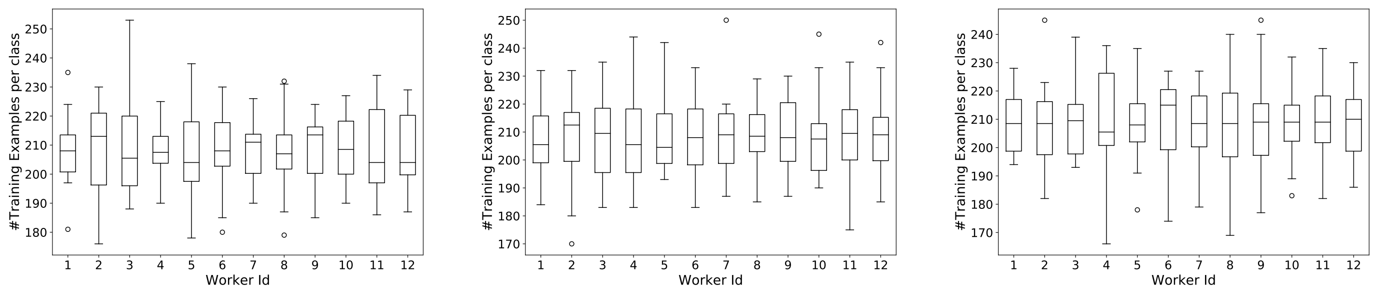
Overall, in the CIFAR-10 series of experiments, the baseline random approach presents nearly the greatest variance, which We will further discuss in section IV-D2. Greater variance in the validation metrics is only met when using a large number of neighbourhoods compared to the number of classes in the dataset. This effect is discussed in the next subsection (IV-C2).

2) ***Effect of Sparsely Populated Neighbourhoods.***: Another interesting insight comes from examining how the model metrics are affected when increasing the number of classes provided in the distribution aware algorithm. The training loss and its variance are constantly decreasing with the increase of the proposed number of neighbourhoods. However, in the validation metrics of Table II, we notice that if we provide the algorithm with a large number of neighbourhoods, their variance becomes larger, indicating that the model might slightly overfit. Since sparsely populated neighbourhoods are broadcasted to all workers, it is more likely that a greater number of examples will be reused from all workers while increasing the number of neighbourhoods. For example, in the case of 40 neighbourhoods, a more closer look on each distinct run asserted the above statement. During one of the runs, as shown in Table III, 11 sparsely populated neighbourhoods appear to slightly harm the validation accuracy which concluded in a value of 0.9330. On the other hand, when 7 sparsely populated neighbourhoods occurred, the validation accuracy managed to reach 0.9370. Note that the validation loss on this case was 0.4329 leading to a 2% enhancement from the results of the baseline method. Overall, while the distribution aware sharding can enhance the value of the metrics, we should not examine a large number of neighbourhoods to avoid the case of multiple sparsely populated ones.

D. Coarse - Grain CIFAR-100

1) ***Metrics and Variance.***: Table IV presents the final training and the validation metrics of training the Resnet-56v1 network of the CIFAR-100 dataset with the coarse - grain labels, having the data sharded to workers with all the aforementioned techniques. Stratified sharding led to results that minimize the training metrics and the variance of the validation metrics. Specifically, the variance of the validation loss and accuracy was $1.65X$ and $6.11X$ smaller than the baseline. Training loss was also minimized by 2%.

While stratified sharding manages to minimize the variance, the actual value of the validation metrics is minimized when using the distribution aware sharding algorithm. In IV-C2 the experimental evaluation indicated that distribution aware sharding, when given the appropriate number of clusters as input, could provide the model with the best validation metrics and smaller variance than the baseline. Table IV can further support this claim. Providing the algorithm with twice the number of classes, appear to have an enhanced validation loss with $1.21X$ less variance from the round - robin sharding. The same applies for the validation accuracy, which meets its best value of 0.806237 in this setup. Note that validation loss is minimized when the number of clusters in the algorithm



(a) Run #1 (b) Run #2 (c) Run #3

Fig. 7: Box plots representing the number of class examples in each shard created randomly for 3 of the runs.

TABLE V: Statistics (5 runs) of final Training and Validation Loss / Accuracy on the Fine CIFAR-100 dataset per method

Method			**Training Metrics**				**Validation Metrics**			
			Loss		Accuracy		Loss		Accuracy	
			Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
Random			0.512164	0.009550	0.744448	0.007855	1.841267	0.026051	0.704707	0.002222
Stratified			0.516125	0.009296	0.744203	0.003107	**1.817435**	**0.011633**	**0.708432**	**0.001225**
Distribution Aware	Class Number	50	0.513846	**0.004085**	0.746785	**0.000764**	1.835961	0.019680	0.707048	0.004526
		100	0.521913	0.008164	0.743808	0.005842	1.851682	0.015945	0.703488	0.003561
		200	**0.510340**	0.006247	**0.746949**	0.001134	1.839176	0.013421	0.708234	0.002264

is set to three times the number of classes (60). However, the variance in this setup appears to suffer from the sparse neighbourhoods effect discussed in IV-C2.

*2) **Random Sharding Weakness.**:* Having discussed the variance minimization that can be achieved from the stratified sharding algorithm, it is important to further understand why the default random approach results to larger variance in the validation metrics. Figure 7 presents a group of box plots for 3 of the runs of the random sharding approach. Each box plot describes the number of training examples from each class that is assigned to each worker. CIFAR-100 consists of 20 equally populated coarse grain labels. Sharding the data set into 12 workers should provide each with approximately 208 training examples from each category. Most workers have a median number of examples per class close to this value and overall $(200, 215)$ as a 50% confidence interval in most cases. However, the box plots indicate that several classes are distributed unequally to the workers. For instance, a closer look on the box plots referring to the second and third runs (Figures 7b and 7c) indicates that most of the workers have $180-230$ and $175-235$ training examples from each category respectively, leading to unbalanced sharding for some classes. Thus, a worker will try to adjust the model more on one specific class leading to greater variance in both training and validation metrics between the training attempts. Of course, an outlier number of examples per class could further hurt the variance, as for instance in the case of workers 2 and 7 from the second run (Figure 7b) and worker 2 from the third run (Figure 7c), since the divergence of the class size compared to the rest will further create dominant or subdominant classes.

This non - uniform view each worker has on the data, is not met on the proposed algorithms. Stratfied shards preserve the percentage of each class size from the whole dataset. Distribution aware technique, considers further hidden stratification, which also avoids this problem, if the appropriate number of clusters is given as input (Section IV-C2).

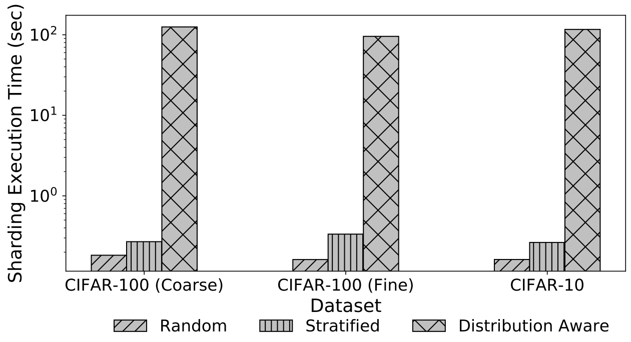


Fig. 8: Average Sharding Technique Time per Dataset

E. Fine - Grain CIFAR100

*1) **Metrics and Variance**:* Table V reveals some interesting findings regarding the validation metrics. To begin with, as discussed in IV-C1 and IV-D1, the variance of the validation metrics is minimized when using stratified shards. Specifically, both validation loss and accuracy are less variant by $2.23X$ and $1.81X$ respectively. Apart from the variance, this series of experiments shows shards derived from the stratified algorithm, also manage to slightly enhance the values of the validation loss and accuracy to $1.81X$ and 70.84% respectively.

As it is shown in Table V training loss is not minimized by shards created from the stratified approach. When distribution aware shards are used, the model appears to minimize the training loss and maximize the training accuracy, followed by similar validation accuracy to the one of the stratified case. While the distribution aware case does not minimize the variance of the validation metrics, it also presents lower values of variance compared to the baseline method.

F. Time Ovehead of Data Sharding

Figure 8 presents the average execution time in seconds for creating shards with technique. For the distribution aware technique we present the mean execution time over all classes examined for each dataset. Stratified and Mod sharding techniques, as same complexity algorithms (see Section III-C), induce almost the same time overhead to the whole training process, which is less than 1 sec. Distribution aware's overhead is ~ 100 seconds, which does not burden the whole training process, since the ResNet-50 network needed approximately 4 hours to converge in the cluster.

Since GPUs are not used to train our models, we need to ensure that the time overhead induced from the sharding algorithms is not important in case accelerators are available. Let us take as an example the family of CIFAR datasets. As we mentioned in IV-B, each of these datasets is used to train a model following the ResNet-50 architecture with the proposed hyperparameters, i.e a global mini-batch size of 128 training examples. Thus, according to II-C, WB is approximately 10 training examples. Benchmarking indicated that TensorFlow needs approximately 0.11 seconds to train a ResNet-56v1 network with a WB mini-batch on a Tesla GPU [49]. In such case, each worker will need 7800 seconds and, therefore, the overhead of the distribution aware approach is insignificant.

G. Discussion

Having presented a detailed evaluation on sharding algorithms, in this section we discuss our findings to help the reader understand the benefits and the drawbacks of each one. Random sharding approach appears to have large variance in the training and validation metrics in general. As we discussed in IV-D2, workers appear to be biased towards one or more classes, which will affect the resulting global model. Stratified sharding comes as a solution to this problem, since each worker has a uniform distribution over the population of each class and, therefore, an equivalent view to the data.

Distribution Aware approach appears to be further useful when the train set presents further hidden stratification (coarse CIFAR-100). It is crucial to set the correct number of neighbourhoods in the algorithm, in order to avoid the sparse neighbourhoods effect (see Section IV-C2). Considering the results we obtained from all the CIFAR family datasets that we examined, we recommend to set the number of neighbourhoods twice the number of classes. However, if prior knowledge indicates no hidden stratification patterns, stratified sharding should be preferred.

To generalize our results, it is important to observe the structure of each dataset. CIFAR-10 contains a few classes (10) with multiple points each (5000). On the contrary, fine-grain CIFAR-100 has multiple classes (100) which are less populated (500 data points / class). Coarse-grain CIFAR-100 is an example of a dataset with hidden patters. These different structures encapsulated by the CIFAR family datasets could allow us to generalize our findings, regarding how each sharding technique affects the variance in the metrics.

In case of applying the Distribution Aware algorithm in much larger datasets, distributed PCA and KMeans should be preferred to minimize the time overhead [50, 51].

V. Related Work

According to our research study, we are the first that examine whether data sharding is able to stabilize the learning process in an asynchronous parameter server setup. However, there are numerous works that attempt to issue other problems in such learning setups. Authors in [52] alter asynchronous SGD to handle stale gradients by including a decaying factor proportionate to a staleness index in the SGD equation. Various other approaches have been also examined regarding staleness as a variable learning rate [53] and aging relating parameters [54]. Furthermore, In a 2020 research work [55], the authors introduce MLfabric, a communication layer that forwards gradients to parameter servers in order to guarantee faster convergence and handle stragglers.

While the aforementioned works focus on solving problems induced by the parameter server architecture itself, our work is inspired from machine learning fundamentals where stratification is used to achieve more stable classification results. For instance, it is widely used in cross validation techniques for standard single node machine learning, where it reduces the metrics' variance [56]. Except core machine learning techniques, stratification is adopted in domain specific model training. Emphasizing on hidden stratification, in [57] they conclude that in some cases, medical related classification models should not be used without considering any hidden patterns. In other domains, stratification is tested over graph partitioning to isolate disjoint subgraphs [58]. Hidden stratification pattern are also used by DSH [59], where PCA and KMeans are adopted to create families of hash functions in order to address the approximate nearest neighbour problem.

In section IV-D2, we actually identified that data skew in workers' shards could increase the variance of the training metrics. In [60], the authors mention that in single node learning data skew can create biased models and propose data generation techniques to overcome this obstacle. While data augmentation was adopted by each worker, in our work we acertained that in the asynchronous distributed setup, more representative shards with no skew lead to less variance in the results. When data are in different locations and data skew problems appear, more complex approaches can be adopted [61]. However, one of our work's core assumptions is that data exist in a shared or distributed file system, which opens the way to exploit simpler approaches as stratification.

VI. Conclusions and Future Work

In this paper, we study the effect of data assignment in an asynchronous distributed learning setup under the parameter server architecture. Specifically, we propose two algorithms, the stratified and the distribution aware one, for assigning training examples on workers. The proposed approaches present smaller variance both on training and validation metrics. Specifically, validation metrics present up to $8X$ and $2X$ less variance in the stratified and distribution aware techniques, followed also by a slight enhancement in the metric values.

Having shown that asynchronous training could be more stable depending on how the data shards are created, we aim to examine how the knowledge of the proposed algorithms could be used in hash functions that would assign data points to workers in real time, supporting online asynchronous learning.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[3] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 6645–6649.

[4] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen *et al.*, "Deep speech 2: end-to-end speech recognition in english and mandarin," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning-Volume 48*, 2016, pp. 173–182.

[5] L. Deng and Y. Liu, *Deep learning in natural language processing*. Springer, 2018.

[6] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 160–167.

[7] E. Kassela, N. Provatas, A. Tsiourvas, I. Konstantinou, and N. Koziris, "Bigoptibase: Big data analytics for base station energy consumption optimization," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 6098–6100.

[8] N. Provatas, E. Kassela, N. Chalvantzis, A. Bakogiannis, I. Giannakopoulos, N. Koziris, and I. Konstantinou, "Selis bda: Big data analytics for the logistics domain," in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 2416–2425.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

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

[11] A. Trask, D. Gilmore, and M. Russell, "Modeling order in neural word embeddings at scale," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37*, 2015, pp. 2266–2275.

[12] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, p. 265–283.

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[14] J. J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. L. Zhang, Y. Wan, Z. Li *et al.*, "Bigdl: A distributed deep learning framework for big data," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 50–60.

[15] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[16] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[17] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[18] A. Castelló, M. F. Dolz, E. S. Quintana-Ortí, and J. Duato, "Analysis of model parallelism for distributed neural networks," in *Proceedings of the 26th European MPI Users' Group Meeting*, 2019, pp. 1–10.

[19] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, and K. Yang, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, p. 1223–1231.

[20] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, "Measuring the effects of data parallelism on neural network training," *Journal of Machine Learning Research*, vol. 20, no. 112, pp. 1–49, 2019.

[21] "Measuring the Limits of Data Parallel Training for Neural Networks." [Online]. Available: http://ai.googleblog.com/2019/03/measuring-limits-of-data-parallel.html

[22] "Distributed communication package - torch.distributed — PyTorch 1.6.0 documentation." [Online]. Available: https://pytorch.org/docs/stable/distributed.html

[23] "MultiWorkerMirroredStrategy." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/MultiWorkerMirroredStrategy

[24] [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/ParameterServerStrategy?hl=el

[25] E. Kassela, N. Provatas, I. Konstantinou, A. Floratou, and N. Koziris, "General-purpose vs specialized data analytics systems: A game of ml & sql thrones," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019.

[26] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in neural information processing systems*, 2011, p. 693–701.

[27] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-sgd for distributed deep learning," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 2016, p. 2350–2356.

[28] N. Provatas, "Exploiting data distribution in distributed learning of deep classification models under the parameter server architecture." in *Proceedings of the VLDB 2021 PhD Workshop (VLDB-PhD 2021), Copenhagen, Denmark, August 16, 2021*, vol. 2971. ceur-ws.org, 2021.

[29] M. Maniruzzaman, M. J. Rahman, M. Al-MehediHasan, H. S. Suri, M. M. Abedin, A. El-Baz, and J. S. Suri, "Accurate diabetes risk stratification using machine learning: role of missing value and outliers," *Journal of medical systems*, vol. 42, no. 5, p. 92, 2018.

[30] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[31] Y. Bengio, "Rmsprop and equilibrated adaptive learning rates for non-convex optimization," *corr abs/1502.04390*, 2015.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[33] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, 2013, p. 1139–1147.

[34] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, 2010, p. 2595–2603.

[35] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola, "Parameter server for distributed machine learning," in *Big Learning NIPS Workshop*, vol. 6, 2013, p. 2.

[36] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 583–598.

[37] A. Smola and S. Narayanamurthy, "An architecture for parallel topic models," *Proceedings of the VLDB Endowment*, vol. 3, no. 1–2, p. 703–710, 2010.

[38] R. Caruana, S. Lawrence, and C. L. Giles, "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping," in *Advances in neural information processing systems*, 2001, p. 402–408.

[39] P. Koch, O. Golovidov, S. Gardner, B. Wujek, J. Griffin, and Y. Xu, "Autotune: A derivative-free optimization framework for hyperparameter tuning," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, Jul 2018, p. 443–452. [Online]. Available: https://dl.acm.org/doi/10.1145/3219819.3219837

[40] S. Gupta, W. Zhang, and F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, Dec 2016, p. 171–180. [Online]. Available: http://ieeexplore.ieee.org/document/7837841/

[41] [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/data/Dataset#shard

[42] S. Lloyd, "Least squares quantization in pcm," *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[43] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA, 1967, pp. 281–297.

[44] K. P. F.R.S., “On lines and planes of closest fit to systems of points in space,” *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, 1901.

[45] “sklearn.decomposition.incrementalpca¶.” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.IncrementalPCA.html

[46] “sklearn.cluster.kmeans¶.” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

[47] Apache Software Foundation, “Hadoop.” [Online]. Available: https://hadoop.apache.org

[48] A. Krizhevsky, “Learning multiple layers of features from tiny images,” Tech. Rep., 2009.

[49] S. Shi, Q. Wang, P. Xu, and X. Chu, “Benchmarking state-of-the-art deep learning software tools,” in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, 2016, pp. 99–104.

[50] “Dimensionality reduction - rdd-based api.” [Online]. Available: https://spark.apache.org/docs/latest/mllib-dimensionality-reduction.html#principal-component-analysis-pca

[51] “Clustering.” [Online]. Available: https://spark.apache.org/docs/3.1.2/ml-clustering.html#k-means

[52] G. Damaskinos, R. Guerraoui, R. Patra, M. Taziki *et al.*, “Asynchronous byzantine machine learning (the case of sgd),” in *International Conference on Machine Learning*. PMLR, 2018, pp. 1145–1154.

[53] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar, “Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd,” in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2018, pp. 803–812.

[54] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, and J. Cheng, “Flexps: Flexible parallelism control in parameter server architecture,” *Proc. VLDB Endow.*, vol. 11, no. 5, p. 566–579, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3177732.3177734

[55] R. Viswanathan, A. Balasubramanian, and A. Akella, “Network-accelerated distributed machine learning for multi-tenant settings,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 447–461.

[56] K. Sechidis, G. Tsoumakas, and I. Vlahavas, “On the stratification of multi-label data,” in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2011, pp. 145–158.

[57] L. Oakden-Rayner, J. Dunnmon, G. Carneiro, and C. Ré, “Hidden stratification causes clinically meaningful failures in machine learning for medical imaging,” in *Proceedings of the ACM Conference on Health, Inference, and Learning*, 2020, pp. 151–159.

[58] J. Nishimura and J. Ugander, “Restreaming graph partitioning: simple versatile algorithms for advanced balancing,” in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013, pp. 1106–1114.

[59] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi, “Dsh: data sensitive hashing for high-dimensional k-nnsearch,” in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 1127–1138.

[60] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, “Data lifecycle challenges in production machine learning: a survey,” *ACM SIGMOD Record*, vol. 47, no. 2, pp. 17–28, 2018.

[61] K. Hsieh, A. Phanishayee, O. Mutlu, and P. Gibbons, “The non-iid data quagmire of decentralized machine learning,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 4387–4398.