Courier: Real-Time Optimal Batch Size Prediction for Latency SLOs in BigDL

Diego Albo Martínez
D.AlboMartinez@student.tudelft.nl
Student, Master Computer Science
Delft University of Technology
Delft, Netherlands

Sharwin Bobde
S.Bobde-1@student.tudelft.nl
Student, Master Computer Science
Delft University of Technology
Delft, Netherlands

Tomasz Motyka
T.T.Motyka@student.tudelft.nl
Student, Master Computer Science
Delft University of Technology
Delft, Netherlands

Lydia Chen
Y.Chen-10@tudelft.nl
Associate Professor
Delft University of Technology
Delft, Netherlands

ABSTRACT

Distributed machine learning has seen immense rise in popularity in recent years. Many companies and universities are utilizing computational clusters to train and run machine learning models. Unfortunately, operating such a cluster imposes large costs. It is therefore crucial to attain as high system utilization as possible. Moreover, those who offer computational clusters as a service, apart from keeping high utilization, also have to meet the required Service Level Agreements (SLAs) for the system response time. This becomes increasingly more complex in multitenant scenarios, where the time dedicated to each task has to be limited to achieve fairness. In this work, we analyze how different parameters of the machine learning job influence the response time as well as system utilization and propose Courier. Courier is a model that, based on the type of machine learning job, can select a batch size such that the response time adheres to the Service Level Objectives (SLOs) specified, while also rendering the highest possible accuracy. We gather the data by conducting real-world experiments on a BigDL cluster. Later on, we study the influence of the factors and build several predictive models which lead us to the proposed Courier model.

KEYWORDS

Deep Learning, Distributed Systems, Hyperparameter optimization, Resource Management, Provisioning, Scheduling

ACM Reference Format:

Diego Albo Martínez, Sharwin Bobde, Tomasz Motyka, and Lydia Chen. 2021. Courier: Real-Time Optimal Batch Size Prediction for Latency SLOs in BigDL. In *Proceedings of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3427921.3450233

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '21, April 19–23, 2021, Virtual Event, France
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8194-9/21/04…$15.00
https://doi.org/10.1145/3427921.3450233

1 INTRODUCTION

Recent years have seen an unprecedented growth of both machine learning (ML) research and applications incorporating machine learning techniques. A natural consequence of such popularization of ML is the rapid growth of data collection. Training datasets for the most advanced ML applications can reach the size of terabytes. The longer time of training ML models naturally steered engineers towards using distributed systems for an increase of parallelization and the total amount of I/O bandwidth.

One of the biggest issues with distributed machine learning is how to effectively compute the gradient among the participating nodes. Currently, synchronous strategies with data parallelism, such as the Synchronous Stochastic Gradient Descent (S-SGD), are widely utilized in distributed training of Deep Neural Networks (DNNs). Its popularity is due mainly to easy implementation yet promising performance. However, such an approach imposes few additional constraints, in comparison with a single server ML, that the engineers have to take into consideration. In systems where the workers have different computational capabilities, the synchronization forces them to wait for each other to move the computation forward, which leads to higher training (response) time. Another thing to consider is how often the synchronization of gradients needs to happen. Gradient computation happens after each batch and so the frequency of synchronization is highly dependent on the batch size.

Another aspect to take into account is that usually distributed ML systems are shared between different jobs. Given that the operational cost of such a system is quite high, one would like to achieve the highest system utilization, resulting in lower operational costs. This is most vital when talking about GPU clusters, being these costly devices and often difficult to utilize to their full extent. However, there exist also systems that provide distributed deep learning capabilities with different architectures, such as BigDL [4], in which it is not so clear how resource usage, parallelism, and other factors affect the response time and overall performance of the system.

In this work, we would like to study how the parameters like batch size, number of concurrent jobs in the system, and the amount of resources given to the job influence the ML job's response time and accuracy in BigDL. Furthermore, we also analyze, particularly

for BigDL, the impact that the load of the system has on I/O waiting time, and its effect on the overall response time.

Our main contributions can be summarized as:

(1) Explore and assess the influence of different models and hyperparameters on response time, accuracy and CPU load in a BigDL cluster.
(2) Evaluate several models using different abstractions to predict the performance of BigDL clusters.
(3) Provide a model combining both response time and accuracy prediction for BigDL jobs, able to compute the optimal batch size at runtime depending of the state of the system and a latency objective with high accuracy, and assess its performance gains.

The remainder of the paper is structured as follows, in Section 2 we summarize previous studies about DNN training in distributed systems and the performance limitations of Spark, on which BigDL relies. In Section 3 we report the configuration used in the experiments and provide and visualize the results obtained, which will provide us with the insights needed in our predictive models described in Section 4. Finally, in Section 5 we describe the *Courier* and evaluate its performance on a simulated scenario.

To guarantee reproducibility and help understanding the results, we provide the code and notebooks used in our GitHub repository: https://github.com/sharwinbobde/Courier-BigDL-Study.

2 BACKGROUND

The rise of popularity of deep learning in recent years has lead to the huge diversification of directions in which one can optimize the performance of deep learning models. Apart from hyper-parameter optimization, researchers were addressing the problem of system parameters optimization to improve the training efficiency. With the introduction of distributed deep learning, the problems of parameter synchronization, efficient communication, and utilization of computing clusters have emerged. We distinguish between the previous works based on the type of optimization they perform.

System parameter optimization. ByteScheduler [20] uses a Bayesian optimization approach. It specifically focuses on auto-tune tensor credit, and partition size for different training models under various networking conditions. ByteScheduler uses auto-tune algorithms to find the optimal system-related configurations.

Hyperparameter optimization. Astra [25] is a framework that exploits the unique repetitiveness and predictability of a deep learning job, to perform an online exploration of the optimization state space, in a work-conserving manner. STRADS [10] exposes parameter schedules and parameter updates as separate functions to be implemented by the user. A parameter schedule identifies a subset of parameters which a given worker should sequentially work on. STRADSAP [11] extends STRADS to a distributed ML setting. AutoKeras [7] enables Bayesian optimization to guide the network morphism for efficient neural network architecture search. The framework develops a neural network kernel and a tree-structured acquisition function optimization algorithm to efficiently explore the search space. The hyperparameter tuning is also offered as a service in some industry-based packages. HyperDrive [21] is a package offered in Azure Machine Learning services, which supports hyperparameter tuning. Amazon SageMaker [16] supports automatic model tuning component that finds the best version of a model by running many training trials on the dataset using the algorithm and a grid of hyperparameters specified by the user.

Combined optimization. Many techniques, however, do not consider the impact that certain hyperparameters and system parameters have on each other. PipeTune [22] strives to optimize both the accuracy and training time of DNNs, while simultaneously tuning the hyper and system parameters. Authors exploit the repetitive pattern of iterative stochastic gradient descent to achieve fast system parameter tuning.

With the datasets being bigger and bigger the need for a distributed approach to deep learning became evident. This approach generates new challenges. In [24] the authors perform comprehensive study of different distributed deep learning frameworks. They show that loading large amounts of data with large size of the batch may become a bottleneck in the case of more shallow networks. Different system architectures for distributed machine learning may give different characteristics. [2] shows that when scaling Parameter Server [14], the throughput fails to scale linearly and that Ring AllReduce [18] achieves better performance due to the efficient use of network bandwidth and overlapping computation and communication.

Communication optimization. The most common architecture for parameter exchange in distributed machine learning is the Parameter Server [14] architecture, which suffers from quite high communication burden as in each step of the algorithm each worker has to send the local gradient update. In [28] the authors proposed a greedy algorithm for dynamically choosing the size of the batch of parameter exchange to minimize the execution time. TicTac [5] derives near-optimal schedules of parameter transfers through critical path analysis on the underlying computational graph. This allows maximal overlap of computation and communication and prevents stragglers arising from random order of parameter transfers at workers. Parallax [12] combines Hyperparameter Server [14] and AllReduce [18] architectures to optimize the amount of data transfers according to the data sparsity. PipeDream [19] combines interbatch pipelining and intrabatch parallelism to improve parallel training throughput, resulting in better overlap of computation with communication. In [29] the authors addressed the problem of load imbalance in Deep Learning clusters resulting in workers waiting for each other to perform weight update in Synchronous Stochastic Gradient Descent, which is the standard algorithm to train distributedly [1, 26]. To alleviate the problem, they proposed a Dynamic Batch Size strategy for DNN training that guarantees the load balance of the cluster during the whole time of training. Such an approach could be useful to take in the context of BigDL.

Resource allocation. Another aspect arising from distributing and parallelizing the computational workloads is how to assign the number of computational resources (CPUs/threads) per worker and how to tune the application parameters to achieve the highest performance. Although the computational frameworks like Apache Spark or BigDL take care of parallelization and scheduling of tasks

they leave the assignment of resources to the end user. This is not so trivial question since the assumption "the higher the number, the better" does not necessarily hold.

In [9] the authors studied how in Apache Spark, depending on the type of application, the number of assigned threads influences the performance. They showed that parts of the application that are I/O bound actually benefit from lower amount of thread and assigning too many threads can hurt the system utilization and hence the performance quite badly. In the context of BigDL, which is built on top of Apache Spark, their findings are consequently applicable. It is not trivial to obtain low response time of distributed training of Deep Neural Networks by simply assigning more computational resources. Similarly to this work, we will model the system utilization by measuring the percentage of CPU usage and total I/O waiting time.

Model-based approach. Having in mind the complexity of distributed deep learning systems, many researchers try the model-based approach in the attempt of describing such systems.

In [15] the authors present a solution to predict training throughput from profiling traces collected from a single-node configuration. In [8] they propose an approach in which they train a deep learning network to predict the execution time for individual parts of a deep learning network that when combined would predict the whole execution time. In [17] they develop a performance model for estimating the throughput of a distributed training job as a function of the number of workers allocated to it. Such a model is used to propose and evaluate heuristics for efficient resource utilization.

Cluster resource optimization. Lastly, many recently published papers in the field of distributed machine learning pursue maximizing the utilization of GPUs during DNN training [13, 23]. In systems of this kind, maximizing the usage of the GPUs can both render a better system performance in terms of throughput, while also meaning higher efficiency and cost-saving. Given the inherent difference between GPU clusters and our BigDL cluster, which relies on CPU computation and the network for shuffling [4], we want to study how desirable the system utilization is in the context of BigDL.

On another note, [23] also introduces the concept of a cluster scheduler that is able to predict and condition the execution of DNN models based on latency SLAs, which we also try to implement here in the context of BigDL.

Performance Modeling in Apache Spark. Since BigDL was implemented on top of Apache Spark, there have been some studies trying to model the performance of different workloads on a Spark Cluster. In [3] and [27] the authors propose and compare multiple machine learning models to predict the response time of jobs. They do this by considering a handful of system-related features, and do not take into account multi-tenancy, a variable number of tasks, or the usage of task-specific parameters in the optimization process.

In this paper, we characterize the performance of Deep Learning jobs in BigDL and analyze different models and their ability to predict the performance of the jobs. Unlike previous literature, which focused on modeling the performance of Deep Learning workflows in GPU clusters on one side, or plain Spark jobs on the other, we tackle the modeling problem of running Deep Learning workflows in a multitenant Spark cluster.

3 EXPERIMENTS

Motivated by the previous studies described in Section 2, we decided to study 4 different factors in our experiments:

1. **CPU number.** The total amount of CPUs assigned to a single job. In Spark terminology, this is the same as the number of Spark executors assigned to a Spark Job.
2. **Batch size.** Batch size used for training the network.
3. **Number of jobs.** Number of concurrent jobs running in the system. These will all run with the same number of CPUs.
4. **Network topology.** Topology of the deep neural network that is being trained.

Factors	**Levels**			
Num. of CPU	1	2	4	8
Batch-size	64	128	256	512
Num. of jobs	1	3	5	
Topology	shallow	deep		

Table 1: Levels of each factor

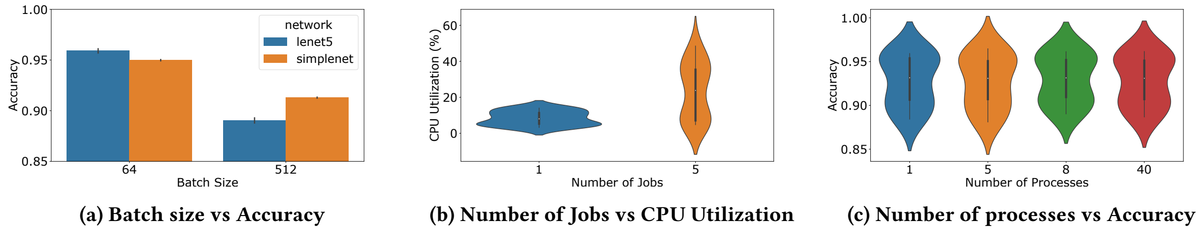
As for the cluster used to conduct the experiments, we used four nodes hosted on Google Cloud Platform (GCP), one master and three workers. For the master node, we chose an e2-standard-8 instance, with 8 vCPUs and 32 GB of memory. As for the three workers, we used e2-standard-16 instances, with 16 vCPUs and 64 GB of memory each. This makes our cluster amount to a total of 48 vCPUs (which in our system will be able to host 48 Spark executors) and 192 GB of memory. We also allocated 3 GB of memory to each Spark executor to be able to fit in all the larger experiments.

3.1 Methodology

To explore the significance and the scale of the interaction between factors we will employ the ANOVA analysis [6]. ANOVA stands for *Analysis of Variance*, and it is a frequently used tool when designing experiments and analyzing their results. ANOVA allows us to compare how multiple factors (i.e. CPU, batch size...) with multiple levels or values each, as well as their interactions, affect the outcome variables of the experiments, namely response time and accuracy in our case.

From the different kinds of experiment designs, we chose to perform two, the 2^k factorial and the Full-Factorial design [6]. The 2^k-factorial makes reference to the binary character of the design. In this design, two values are chosen for each of the factors, with these normally being the extremes of the search interval for that factor. This limits the number of experiments to 2^k, with k being the number of factors. This allows us to get a grasp of the overall behavior of the system with a limited number of experiments.

For a more in-depth analysis, we conduct the Full-Factorial design. In this design, all levels of all factors are combined, thus resulting in a higher number of experiments, but providing a more comprehensive view of the system performance.



(a) Batch size vs Accuracy (b) Number of Jobs vs CPU Utilization (c) Number of processes vs Accuracy

Figure 1: Plots summarizing the results of the 2^K Experimental Design with the response time as output value

We conducted both 2^k-factorial and full factorial experiments, which were followed by ANOVA analysis. After some initial exploration of the results, we concluded that the accuracy was only affected by the batch size in a significant way, as is shown in Figure 1. Hence, we report the ANOVA results only for the response time. Each experiment was replicated 3 times. For the full results of the study, covering all the factors, the reader can consult Appendix A. Here, for the sake of simplicity, we present only the most important ones.

3.2 2^k Factorial

First, to get a general perspective of how influential are the considered factors, we performed a 2^k-factorial analysis. Only the extreme levels of each factor were taken into account. Having the replication equal to 3, this resulted in 48 single experiments (N_{2^k}). After collecting the results, we ran the ANOVA analysis, whose results are in Table 2.

$$\begin{aligned} N_{2^k} &= |c| \cdot |b| \cdot |j| \cdot |t| \cdot r \\ &= 2 \cdot 2 \cdot 2 \cdot 2 \cdot 3 \\ &= 48 \ runs \end{aligned}$$

	F	$P>$F	t	$P>\|t\|$	[0.025	0.975]
cpu	7.001	1.159e-02	0.449	0.656	-10.198	16.031
batch	9.063	4.50e-03	-3.204	0.003	-0.530	-0.120
njobs	-0.004	1	0.344	0.733	-17.208	24.265
network	429.306	5.35e-23	-6.165	0	-302.834	-153.301

Table 2: ANOVA analysis 2^k factorial.

By looking at only first order main factors, we can see that only the *batch-size* and the *network* present a confidence interval that does not include zero and thus presents significant variation between the levels. That said, we provide further insights with more levels per factor in the following analyses.

3.3 Full Factorial

Next, we conducted a full factorial experiment. Having 4 factors with 4, 4, 3 and 2 levels respectively and the replication factor equal to 3, it required 288 single experiments (N_{full}). After obtaining the results, we did the ANOVA analysis, whose results are in Table 3.

$$\begin{aligned} N_{full} &= |c| \cdot |b| \cdot |j| \cdot |t| \cdot r \\ &= 4 \cdot 4 \cdot 3 \cdot 2 \cdot 3 \\ &= 288 \ runs \end{aligned}$$

	F	$P>$F	t	$P>\|t\|$	[0.025	0.975]
cpu	20.449	9.92e-06	-1.248	0.213	-30.170	6.768
batch	467.919	8.95e-57	-4.773	0	-0.987	-0.410
njobs	42.765	4.12e-10	-0.253	0.801	-28.124	21.728
network	5.108	2.47e-02	0.470	0.639	-64.841	105.437
cpu:njobs	15.7896	9.55e-05	4.634	0	7.309	18.124

Table 3: ANOVA analysis full factorial.

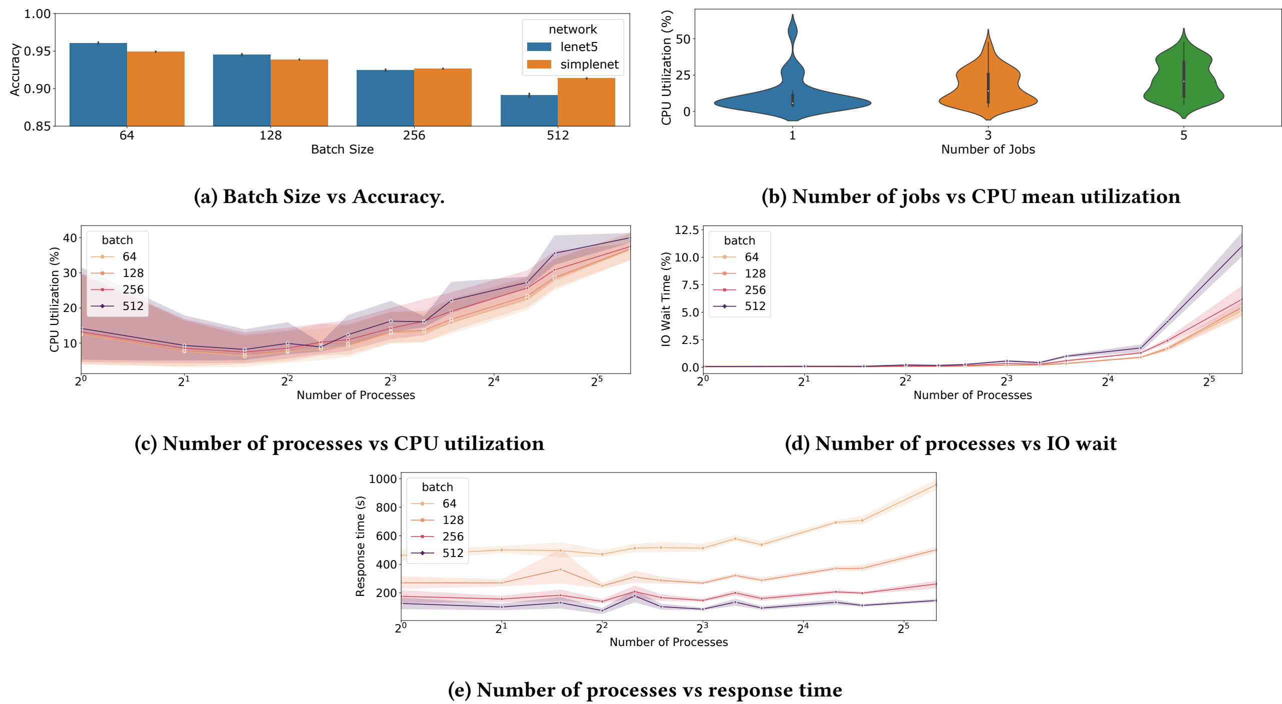
It can be seen that only the *batch-size* and the combination *cpu:njobs* have values of $P > |t|$ smaller than 0.05. This means that at the 95% confidence level only those factors have significant differentiating power. We also extract multiple insights from the full experiment data, these we summarize in Figure 2. From now on in the paper we will refer to the combination of the *cpu* and *njobs* factors as the *processes* factor, since it will be a constantly used feature in our models.

In Figure 2a we can observe the same as we saw in the 2^k factorial design, the batch size and the network are the two main predictors for the accuracy. In terms of system utilization, we see in Figures 2b and 2c that the CPU usage is mainly influenced by the interaction of the number of CPUs and the number of jobs. This also results in an increase in the time spent waiting for IO, as seen in Figure 2d, which can be explained by the more extensive synchronization with many Spark executors in the job, and for the contention for disk resources with more jobs in the system. We will use these insights to model the jobs in an analytical way using operational laws in Section 4.2.

Finally, we also relate the number of processes, and thus the CPU utilization, to the response time in Figure 2e. Showing that in the case of BigDL, workflows with smaller batch sizes scale much worse than bigger batches. This can be due to the highest number of synchronization steps for small batch sizes, which in the presence of higher contention for resources like disk or network I/O, can result in lengthy delays in execution time.

4 PREDICTIVE MODELS

After getting the results from the ANOVA analysis and understanding the relationship and dependencies within the data, we analyze



(a) Batch Size vs Accuracy.

(b) Number of jobs vs CPU mean utilization

(c) Number of processes vs CPU utilization

(d) Number of processes vs IO wait

(e) Number of processes vs response time

Figure 2: Plots summarizing the results of the Full Factorial Experimental Design

the system in several ways. For that, we build three different models that tackle the modeling problem using different approaches. First, we use the model derived from the ANOVA analysis to assess how well the interactions between the factors are able to estimate the response time of the system based on the characteristics of the job. The model resulting from the ANOVA analysis simply represents a linear model in which each of the factors and each of their interactions is given a weight, and the output is the linear combination of the weighted factors.

Next, we model the system by dividing it into two main resources: CPU and Disk. We then employ operational laws and queuing theory to estimate the response time with the help of this abstraction. Finally, we compare these previous approaches with a more refined one, we use a more complex regressor, such as a Random Forest, and analyze how well we are able to predict the response time and the accuracy based on the job parameters.

To assess the performance of the different models we will employ two metrics. The Mean Squared Error (MSE) will help us determine the approximate error that the model incurs in on average. However, given the considerable difference in feature scaling—accuracy is between 0 and 1 and time can reach values over a thousand—we will use also the R^2 score to give a general idea of the quality of the fit. This will disregard the difference in scaling and provide a clearer picture of how well each output label is predicted.

To determine the performance of the model, we divide the data gathered from the experiments into two distinct datasets: the training set, comprising 80% of the data, and a test set composed of the remaining 20%. For hyperparameter tuning, when necessary, we will train a grid search 5-fold cross-validation on the training set data to choose the best hyperparameters to predict each of the output labels.

4.1 ANOVA linear model

The first model that we use to try to estimate the results of the experiments (both accuracy and response time) is the model resulting from the ANOVA analysis explained in previous sections. We use the model of the Full Factorial analysis with the four factors—batch size, CPU, *njobs* and network—and all their interactions. This leaves us a model with 15 coefficients plus the intercept, which we fit to the training dataset. We predict the accuracy and response time for the data points in the test set and calculate several metrics such as the MSE and the R^2 score for both output variables of the model. The experiment results can be seen summarized in Table 4.

Label	MSE	R^2
Accuracy	1.9252×10^{-5}	0.95726
Response Time	9928.175	0.64005

Table 4: Mean Squared Error and R^2 score for each of the ANOVA model on the testing data

Despite accounting for all factors and their interactions, the fit of the model is not great at just a 64% R^2 score, and it proves to be the worst model of all tested in terms of fitting the response time. The accuracy is better predicted by the model; however, as seen in previous sections, the accuracy is an easier output to predict given

that it is determined mainly by just the batch size and the network, as opposed to the response time which shows much more intricate interactions.

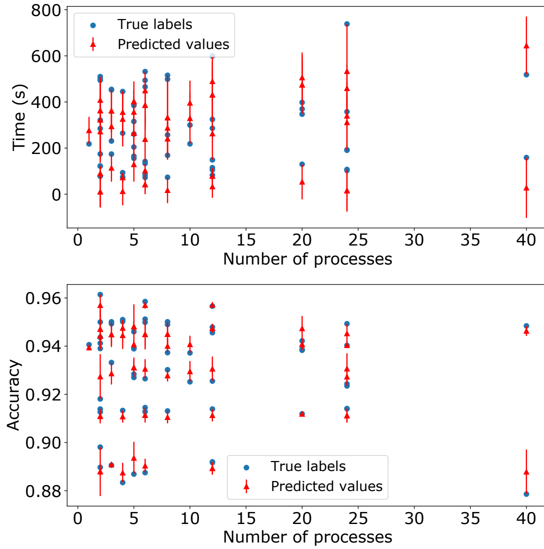


Figure 3: Comparison of the actual and predicted values for Response Time (top) and Accuracy (bottom) with the ANOVA model

4.2 Operational Laws & Queuing Theory

The insights gathered from the ANOVA analysis back in Section 3 lead us to think that the system could be modeled by means of two main abstractions, CPU and Disk. The training process of a BigDL job is comprised of clear stages. Once the job has started, the workers continuously have to fetch data from the disk, train on the data from the batch, communicate the results and updates, and repeat the cycle again. We take the two main tasks from this cycle and model them as a queuing network made up of a CPU and a disk. For a higher level of abstraction, we model the CPU as an M/M/1/1/PS[1] queue, since a server has multiple CPUs and no more jobs than free CPUs will be scheduled in a worker. This means that all jobs present in a server at the same time can be executed in the CPU concurrently.

On the other hand, we take the disk as the main bottleneck in the system and model it as an M/M/1 queue. Jobs arrive at this queue with an arrival rate depending on the number of jobs currently being executed on the cluster. In the disk, the jobs have to retrieve the appropriate chunk of data needed for the next batch, which leads us to model the service rate of the disk as a function of the batch size and the number of processes. The main structure of the model can be seen in Figure 4.

We can then model the response time of the full system as

[1]This is expressed in Kendall's notation. This refers to a queue with an arrival and service processes following exponential distributions, a number of servers equal to 1, with no queue, which follows a Processor Sharing (PS) scheduling policy

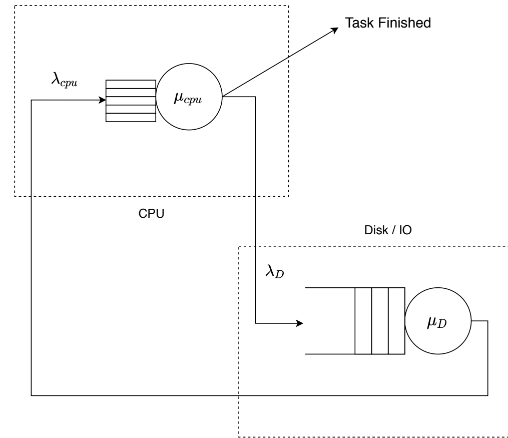


Figure 4: Modeling of the system based on queues

$$\begin{aligned}\mathbb{E}[T] &= \frac{N}{B} \times N_{epoch} \left(\mathbb{E}[S_{cpu}] + \mathbb{E}[T_D]\right) \\ &= N_T \left(\mathbb{E}[S_{cpu}] + \mathbb{E}[T_D]\right)\end{aligned}$$

Where N is the total number of samples in the dataset, B is the batch size, N_{epoch} is the number of epochs in the training job, and those are multiplied by the service time of the CPU (since the CPU has no queue) and the response time of the disk. Intuitively, we can substitute the first part of the equation by a constant that only depends on the batch size, since the size of the dataset and the number of epochs stay constant in the tests. This constant symbolizes the number of times that the job will have to go through the cycle of fetching data from the disk and training on it.

4.2.1 Modeling the CPU. We start by modeling the CPU, more specifically we need to estimate a function for the expected service time $\mathbb{E}[S_{cpu}]$ of the CPU to then estimate the arrival rate for the disk. To do so, we check the resulting logs from running the BigDL jobs on the cluster and analyze for each batch size the throughput of the system. The results obtained are shown in Table 5.

Batch	Throughput (records/s)	$\mathbb{E}(S)$
64	1600	0.04
128	3300	0.0383
256	5700	0.0449
512	10200	0.0502

Table 5: Average throughput seen with different batch sizes

As can be seen from the results, the throughput in terms of records per second increases with the size of the batch, however the service time per record tends to stay the same. For the sake of simplicity, we model the CPU service time as a constant service time of 0.045 seconds per record. We see that no matter the arrival rate of records to the CPU, the service time stays constant, leading us to calculate the arrival rate to the disk as

$$\lambda_D = \frac{1}{\mathbb{E}[S_{cpu}]} = 22.\overline{22}\ req/s$$

However, this is just for one job running concurrently in the system. Given the nature of the CPU, many jobs can be executing at the same time, thus making the arrival rate at the disk scale with the number of tasks in one worker linearly. This number can vary due to a bigger number of spark executors (cpus) in the system, or also because of an increased number of jobs in the system. We approximate the average of the tasks running in one server as the total number of processes (cpu × njobs) divided by the number of workers, and limit the minimum amount to 1.

$$\lambda_D = \frac{N_{proc}}{\mathbb{E}[S_{cpu}]} = max(1, \frac{N_{cpu} \times N_{jobs}}{N_w}) \times \mu_{cpu}$$

4.2.2 Modeling the Disk. Now that we have an idea about the service rate of the CPU and the arrival rate at the disk, we can isolate the service rate in the main equation to estimate the service rate of the disk, μ_D.

$$\begin{aligned}\mathbb{E}[T] &= N_T\left(\mathbb{E}[S_{cpu}] + \mathbb{E}[T_D]\right) \\ &= N_T\left(\mathbb{E}[S_{cpu}] + \frac{1}{\lambda_D - \mu_D}\right)\end{aligned}$$

$$\hat{\mu}_D = \frac{1}{\mathbb{E}[T]/N_T - \mathbb{E}[S_{cpu}]} + \lambda_D$$

To get an estimate of the function behind μ_D we substitute the values in the equation of $\mathbb{E}[T]$, N_T and λ_D for each data point in the training set, and calculate the μ for it. We then have to find a function that given the batch and the number of processes can estimate the service rate of the disk. We do this with a 2D linear regression using the batch and number of processes. Now, we are able to predict for each task the expected service rate of the disk, and with that, using the formula for the response time, we get an estimate of the $\mathbb{E}[T]$ of the system.

Label	MSE	R^2
Response Time	4538.05	0.8330

Table 6: Mean Squared Error and R^2 score for each of the regressors on the testing data

We evaluate the fit of the model again by computing the MSE of the test set and the R^2 score of the model in Table 6. Interestingly, this model based on queuing theory reports an overall lower MSE and a better fit than the linear ANOVA model studied in the previous section.

4.3 Random Forest Regression

Finally, we try to model the behavior of the system using a more sophisticated regressor. To choose the algorithm for this task, we used several regression algorithms from the `sklearn` library, namely, the Linear, Lasso and Ridge Regression from the linear models, and the Decision Tree and Random Forest regressor from among the non-linear models. Random Forest ended up being the one that reported the minimum MSE on both the accuracy and time prediction task by a wide margin.

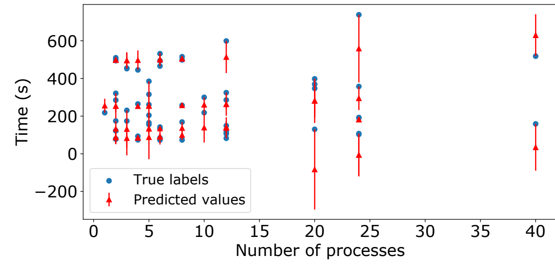


Figure 5: Comparison of the actual and predicted values for Response Time.

4.3.1 Feature Selection & Training. Following the factor significance analysis conducted in Section 3, we chose to train the forest with 5 features: batch size, CPU number, number of jobs in the system, network type, and total processes, which is a multiplication of CPUs and njobs, which has proved to be a good predictor of the response time.

For the training and evaluation process, we standardize the data points and cross-validate the model with 5 folds in a grid search to choose the best combination of the parameters in Table 7. We do this using both the response time of the system and the accuracy as output labels, with the goal of being able to predict both with a low error margin.

Hyper-Parameter	Values	RF-Accuracy	RF-Time
Num. Estimators	{50 ...2000}	150	2000
Max. Features	{auto, log, sqrt}	auto	auto
Max Depth	(4, 10)	9	6
Criterion	mse, mae	mae	mae

Table 7: Hyperparameters chosen for each of the Random Forest Regressors

4.3.2 Performance Evaluation. After this we measure the performance of the fitted classifiers on the test set using the same metrics as with the previous models. The results can be seen in Table 8. The Random Forest is much better than the one reported in previous sections with other models in both accuracy and response time prediction, being the latter an order of magnitude better than the next closest model.

To get a glimpse at how well the model is fitting the data points from the test set, we plot the true and predicted labels for both the response time and accuracy in Figure 6. As the plot shows, the model is able to predict with low error the response time and accuracy no matter the number of processes in the system. Moreover, we plot the feature importances as given by each of the models in Figure 7.

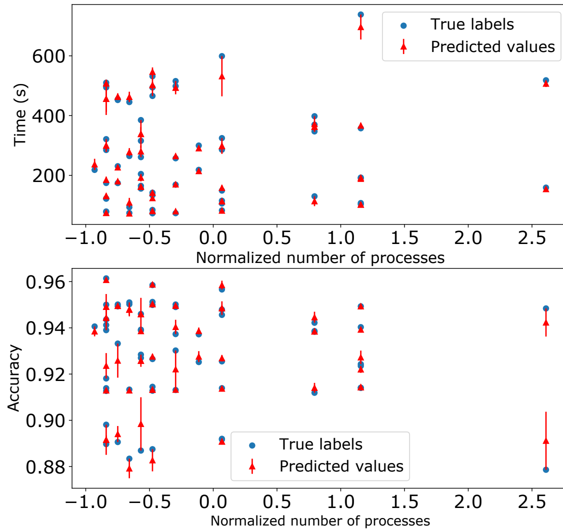


Figure 6: Comparison of the actual and predicted values for Response Time (top) and Accuracy (bottom)

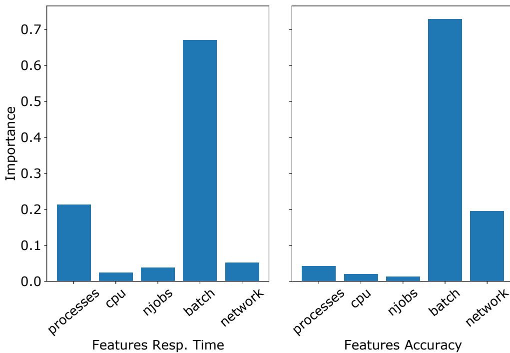


Figure 7: Importance of the features to predict Response Time (left) and Accuracy (right)

Label	MSE	R^2
Accuracy	1.3590×10^{-5}	0.9711
Response Time	376.659	0.9861

Table 8: Mean Squared Error and R^2 score for each of the regressors on the testing data

Indeed, the *processes* feature proves again the same we could see in the ANOVA test, that it is the second most reliable feature to predict the response time. As for the accuracy, we observe what was expected, being the batch size and the network being the main variables that determine the final output.

One more thing to consider is, when should the classifier be retrained. In these experiments, we did consider homogeneous compute nodes. Adding a few more nodes of *similar* compute power should not affect the results much, but adding a few more compute nodes in combination with a single vCPU will lead to performance degradation. Although, adding several more homogeneous nodes will also require retraining because the communication overhead will increase and many more things will be needed to be taken into consideration to optimize the performance.

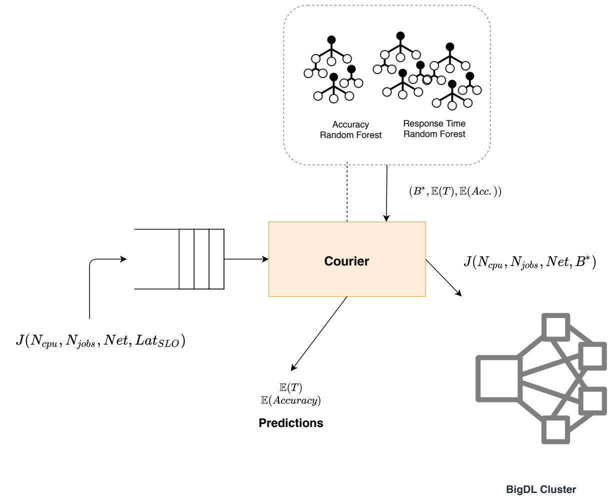


Figure 8: Architecture of Courier and Pipeline for Job Scheduling

5 OPTIMIZATION STRATEGY

5.1 Courier Description

After demonstrating the good predicting performance of the Random Forest Regressor, we incorporate both models into a single overall system able to predict the accuracy and response time based on the input parameters of the job, Courier. With Courier, we combine the capabilities of predicting the response time and accuracy of a job, and use both predictions to estimate the optimal batch size given a latency SLO specified by the user. We provide a high level description of what Courier does in Figure 8.

5.2 Description of Experiments

To test the performance improvements derived from the use of Courier, we test its performance in the same cluster used to evaluate the experiments in Section 3. We compare its performance against two different baselines:

(1) **Naive Scheduler.** In one case, before scheduling a job in the cluster, we input the job characteristics into Courier alongside with a latency SLO to get the optimal batch size to (1) Fulfill the SLO and (2) Achieve maximum accuracy within that time frame. We want Courier to be on average, better than a naive scheduler that chooses the optimal batch

N	CPUs	Njobs	SLO (s)	*Batch**	*Acc.**	*Time** (s)
1	1	5	422	128	93.86%	280.19
2	2	1	412	128	94.44%	299.59
3	4	1	254	128	93.88%	240.56
4	4	2	342	128	94.75%	280.23
5	8	3	166	512	89.27%	112.70

Table 9: Random jobs and Courier's chosen batch and predictions. The values resulting of Courier prediction are marked with an asterisk.

size uniformly from those available, in our case 64, 128, 256, and 512.

(2) **Boosting Model.** Apart from that, we compare the estimations made by Courier against the model used in [3] to predict the runtime of Spark applications. There, the authors employ and compare several regression models, of which they settle on a Gradient Boosting Regression model as it offers the best all-around performance. With this comparison we want to demonstrate the advantage gained from considering the parallelism level of spark applications when making predictions of response time.
Gradient Boosting, like other boosting algorithms, attempts to model a function by using an ensemble of weaker classifiers, most often decision trees. The main concept of these models is iteratively fitting new weak models on the samples that the previous models had a tougher time predicting. In Gradient Boosting, the loss of a learner is minimized by the next learner using a gradient descent algorithm.
In [3], the features used to model the response time of several types of jobs are limited to system parameters such as executor cores, memory and shuffle parameters. To make a fair comparison, we will input the executor cores to the model, since it is the only parameter that we actively change in our experiments from those proposed. We also incorporate the batch size and model type being this application specific parameters that are vital for modeling the performance of the job.

We conduct the experiments by generating randomly the parameters for jobs, such as the number of CPUs, number of jobs, and latency SLO. For the last one, we draw from a normal distribution with mean 300 and standard deviation of 100. To test the generalization of the model, we add extra levels to the number of jobs, which can take now any value between 1 and 6 instead of the three fixed values studied. The five jobs submitted to the cluster as well as the SLOs and the chosen batch size for the job are summarized in Table 9. We replicate the experiments three times and average the results of each job for both Courier and the random batch chooser.

5.3 Experiment Results

5.3.1 Comparison against Naive Scheduler. After running the experiments, we computed several metrics to assess the performance of each of the two models. With the aim of relating the results to those seen beforehand in Section 5, we report the Mean Squared Error (MSE) seen on both the predicted accuracy and the response time. In accordance with the performance of the test set, across all the experiments, Courier reports an *MSE* of 367.58 for the response time and of 4.745×10^{-6} for accuracy.

Apart from that, we also quantify the improvement in terms of satisfying SLO requirements and accuracy improvements when those SLOs are satisfied. We summarize both of these aspects in Figure 9. As proved by Figure 9, Courier is able to on average fulfill the SLO requirements 100% of the time, as opposed to 60% of the baseline. Moreover, as shown in the figure, in those jobs where both fulfill the latency requirement, Courier shows a higher accuracy across the board, further confirming that the choice of batch for the SLO was accurate.

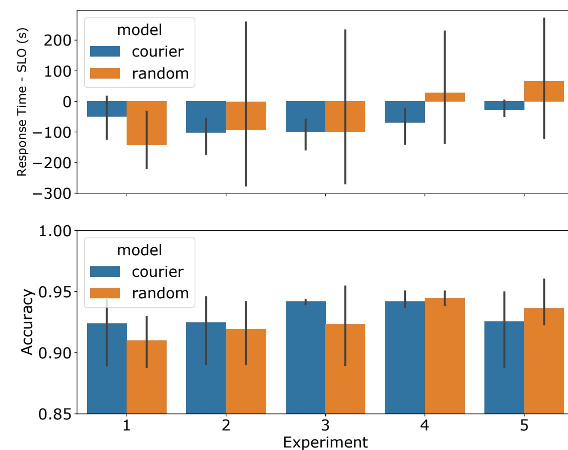


Figure 9: Comparison of the performance of Courier and the Random Batch selector in terms of SLO fulfillment (top) and accuracy (bottom). The experiment number is the same as shown in Table 9.

5.3.2 Comparison against the Gradient Boosting Model [3]. We then compare our model's ability to predict the performance of the training jobs in terms of response time and accuracy when compared to the Gradient Boosting Model. For this, we fit both models on the same data: for Courier we use the features and model parameters described in Section 4.3. For the Gradient Boosting Model we use the same cross-validation technique as for Courier, and tune the parameters accordingly, and end up selecting the parameters as shown in Table 10. As mentioned before, in Courier we use features determining the parallelism level in the cluster, whereas the Boosting model uses only the number of cores as the system feature.

We then compare the performance of both models on the testing data. The results are shown in Table 11. As can be seen, both of the models are able to fit almost identically the expected accuracy, however, Courier outperforms the boosting model by a wide margin in terms of predicting the response time of the job. We can already perceive that having parallelism level as a feature is making a difference.

Hyper-Parameter	Values	RF-Accuracy	RF-Time
Num. Estimators	{50 ... 2000}	50	50
Loss	{ls, lad, huber}	ls	lad
Max Depth	(3, 10)	3	3

Table 10: Hyperparameters chosen for each of the Gradient Boosting Regressors

Finally, we compare the performance on the new set of random experiments as done with the naive scheduler, and compare the error in response time and accuracy predictions. The results can be seen in Figure 10, where we can see that Courier beats the boosting model in all cases when predicting the accuracy, and performs better on average when predicting the response time.

Model	Label	MSE	R^2
Courier	Accuracy	1.3590×10^{-5}	0.9711
	Response Time	376.659	0.9861
Gradient Boost [3]	Accuracy	1.3592×10^{-5}	0.9711
	Response Time	3024.48	0.8887

Table 11: Mean Squared Error and R^2 score for each of the regressors on the testing data

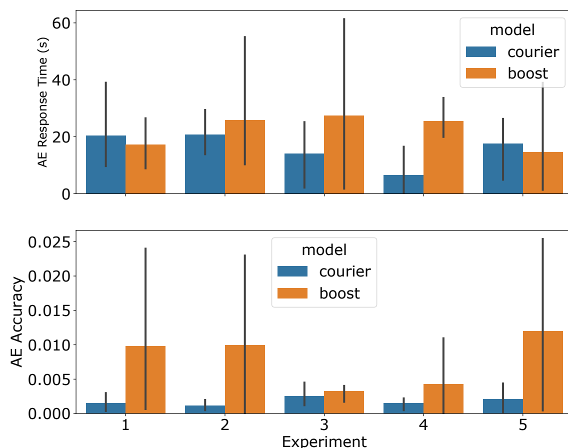


Figure 10: Absolute Error (AE) in Response Time and Accuracy predicted by Courier and the Boosting model

This is further confirmed by analyzing the statistics reported by both models in the new set of experiments, with Courier showing an MSE of 367.58 in correspondence with the results on the testing data, a result more than 50% better than the boosting model, which reports an MSE of 803.26.

5.4 Limitations

As a starting point, we have demonstrated that our Courier model is able to choose the optimal batch for a newly arrived job based on the properties of the job and the desired SLO. However, this being the first prototype, it is lacking some features that would be needed to make Courier a better overall system to manage job scheduling in a BigDL cluster. Below we list some of the desired properties that should be implemented in future work:

(1) **Dinamic number of epochs.** For this initial prototype, we gathered the accuracies and the times reported by each experiment in each of the 10 epochs we ran for each experiment, however at the time of predicting the response time and choosing a batch size we only considered to run the experiment for 10 epochs. A better approach would be to train the model on an extra feature, the number of epochs and the time per epoch. This way, Courier could be able to choose a smaller batch size which, even though slower per epoch, has a higher statistical efficiency and can reach a better accuracy in a shorter time.

(2) **Predictions for Multiplexing Jobs.** Courier could be able to predict the run time of jobs submitted at different points in time running at the same time in the system. This would entail Courier fetching metrics from the actual utilization of the cluster and when a job arrives, calculate the response time of the cluster with the sum of the new job and the already running jobs. This would also entail recomputing the response time predictions and SLOs for the already running jobs, which would effectively increase the complexity of the scheduling.

6 CONCLUSION

In this work, we have analyzed the importance of different features associated with training Neural Networks on BigDL cluster in the context of training time and accuracy. We have shown that the most important parameters to select when scheduling one or multiple DL jobs are the batch-size and the total number of processes. Knowing that, we designed 3 different models, the ANOVA linear model, queuing theory model, and Random Forest Regression model, to describe the dependability of training time on selected features and see how accurate those models will be in predicting the optimal parameters for the job. Seeing that the Random Forest Regressor has the best predicting performance, we implemented Courier, a model that combines two Random Forest Regressors where one is responsible for maximizing the accuracy and the other one for minimizing the response time. Such model selects the job parameters to stay within the latency constrains and also maximize the accuracy.

7 ACKNOWLEDGEMENT

This work has been partly funded by the Swiss National Science Foundation NRP75 Dapprox project 407540_167266.

REFERENCES

[1] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021* (2017).
[2] Salem Alqahtani and Murat Demirbas. 2019. Performance Analysis and Comparison of Distributed Machine Learning Systems.

[3] Zemin Chao, Shengfei Shi, Hong Gao, Jizhou Luo, and Hongzhi Wang. 2018. A gray-box performance model for Apache Spark. *Future Generation Computer Systems* 89 (2018), 58–67.
[4] Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, et al. 2019. Bigdl: A distributed deep learning framework for big data. In *Proceedings of the ACM Symposium on Cloud Computing*. 50–60.
[5] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and R. Campbell. 2019. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. *arXiv: Distributed, Parallel, and Cluster Computing* (2019).
[6] Raj Jain. 2008. *The art of computer systems performance analysis*. john wiley & sons.
[7] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-Keras: An Efficient Neural Architecture Search System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 1946–1956. https://doi.org/10.1145/3292500.3330648
[8] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Mcgough. 2018. Predicting the Computational Cost of Deep Learning Models. 3873–3882. https://doi.org/10.1109/BigData.2018.8622396
[9] Sobhan Omranian Khorasani, Jan S. Rellermeyer, and Dick Epema. 2019. Self-Adaptive Executors for Big Data Processing. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 176–188. https://doi.org/10.1145/3361525.3361545
[10] Jin Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth Gibson, and Eric Xing. 2016. STRADS: a distributed framework for scheduled model parallel machine learning. 1–16. https://doi.org/10.1145/2901318.2901331
[11] Jin Kyu Kim, Abutalib Aghayev, Garth A. Gibson, and Eric P. Xing. 2019. STRADS-AP: Simplifying Distributed Machine Learning Programming without Introducing a New Programming Model. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 207–222. https://www.usenix.org/conference/atc19/presentation/kim-jin
[12] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. 2019. Parallax: Sparsity-Aware Data Parallel Training of Deep Neural Networks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 43, 15 pages. https://doi.org/10.1145/3302424.3303957
[13] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. 2019. CROSSBOW: scaling deep learning with small batch sizes on multi-gpu servers. *arXiv preprint arXiv:1901.02244* (2019).
[14] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 583–598.
[15] Zhuojin Li, Wumo Yan, Marco Paolieri, and Leana Golubchik. 2020. Throughput Prediction of Asynchronous SGD in TensorFlow. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (Edmonton AB, Canada) *(ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 76–87. https://doi.org/10.1145/3358960.3379141
[16] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, Can Balioglu, Saswata Chakravarty, Madhav Jha, Philip Gautier, David Arpin, Tim Januschowski, Valentin Flunkert, Yuyang Wang, Jan Gasthaus, Lorenzo Stella, Syama Rangapuram, David Salinas, Sebastian Schelter, and Alex Smola. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 731–737. https://doi.org/10.1145/3318464.3386126
[17] S. Lin, M. Paolieri, C. Chou, and L. Golubchik. 2018. A Model-Based Approach to Streamlining Distributed Training for Asynchronous SGD. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 306–318. https://doi.org/10.1109/MASCOTS.2018.00037
[18] A.R. Mamidala, Jiuxing Liu, and D.K. Panda. 2004. Efficient Barrier and Allreduce on Infiniband clusters using multicast and adaptive algorithms. 135– 144. https://doi.org/10.1109/CLUSTR.2004.1392611
[19] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646
[20] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 16–29. https://doi.org/10.1145/3341301.3359642
[21] Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. 2017. HyperDrive: Exploring Hyperparameters with POP Scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (Las Vegas, Nevada) *(Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3135974.3135994
[22] Isabelly Rocha, Nathaniel Morris, Lydia Y. Chen, Pascal Felber, Robert Birke, and Valerio Schiavoni. 2020. PipeTune: Pipeline Parallelism of Hyper and System Parameters Tuning for Deep Learning Clusters. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 89–104. https://doi.org/10.1145/3423211.3425692
[23] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 322–337.
[24] Shaohuai Shi and Xiaowen Chu. 2017. Performance Modeling and Evaluation of Distributed Deep Learning Frameworks on GPUs. (11 2017).
[25] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. 2019. Astra: Exploiting Predictability to Optimize Deep Learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 909–923. https://doi.org/10.1145/3297858.3304072
[26] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S Rellermeyer. 2019. A Survey on Distributed Machine Learning. *arXiv preprint arXiv:1912.09789* (2019).
[27] G. Wang, J. Xu, and B. He. 2016. A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 586–593. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0088
[28] Shaoqi Wang, Aidi Pi, and Xiaobo Zhou. 2019. Scalable Distributed DL Training: Batching Communication and Computation. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (Jul. 2019), 5289–5296. https://doi.org/10.1609/aaai.v33i01.33015289
[29] Qing Ye, Yuhao Zhou, Mingjia Shi, Yanan Sun, and Jiancheng Lv. 2020. DBS: Dynamic Batch Size For Distributed Deep Neural Network Training. *arXiv preprint arXiv:2007.11831* (2020).

A APPENDIX

In this Appendix, we present full results of the ANOVA analysis.

	P >F	$P > \|t\|$	[0.025	0.975]
cpu	7.18e-02	0.665	-21.355	33.021
batch	6.28e-14	0.004	-1.074	-0.225
njobs	1.92e-02	0.740	-35.930	50.045
network	6.15e-02	0.389	-88.489	221.496
cpu:njobs	1.48e-01	0.147	-2.042	13.039
cpu:batch	2.38e-02	0.484	-0.100	0.049
batch:njobs	2.41e-01	0.878	-0.127	0.109
cpu:network	5.02e-01	0.467	-37.008	17.367
batch:network	9.21e-01	0.961	-0.435	0.415
njobs:network	9.02e-01	0.544	-55.926	30.049
cpu:batch:njobs	4.70e-01	0.470	-0.028	0.013
cpu:batch:network	4.74e-01	0.859	-0.068	0.081
cpu:njobs:network	4.82e-01	0.331	-3.884	11.196
batch:njobs:network	8.06e-01	0.692	-0.095	0.141
cpu:batch:njobs:network	4.90e-01	0.490	-0.028	0.014

Table 12: ANOVA analysis 2^k factorial.

	P >F	$P > \|t\|$	[0.025	0.975]
cpu	9.92e-06	0.213	-30.170	6.768
batch	8.95e-57	0.000	-0.987	-30.170
njobs	4.12e-10	0.801	-28.124	21.728
network	2.47e-02	0.639	5.108	105.437
cpu:njobs	9.55e-05	0.000	7.309	18.124
cpu:batch	4.22e-03	0.642	-0.055	0.114
batch:njobs	9.39e-04	0.498	-20.449	9.92e-06
cpu:network	1.12e-01	0.711	-21.942	14.997
batch:network	5.01e-01	0.648	-0.222	0.356
njobs:network	9.54e-01	0.943	-24.028	25.824
cpu:batch:njobs	3.20e-03	0.004	-0.045	-0.008
cpu:batch:network	8.95e-01	0.812	-0.070	0.055
cpu:njobs:network	8.12e-01	0.980	-5.475	5.339
batch:njobs:network	8.822e-01	0.804	-0.095	0.074
cpu:batch:njobs:network	8.41e-01	0.842	-0.016	0.020

Table 13: ANOVA analysis full factorial.