2022 30th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)

2022 30th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS) | 978-1-6654-5580-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/MASCOTS56607.2022.00011

# DTS: A Simulator to Estimate the Training Time of Distributed Deep Neural Networks

Wilfredo J. Robinson M.
*Computer Science Department*
Saint Louis University, USA

Flavio Esposito
*Computer Science Department*
Saint Louis University, USA

Maria A. Zuluaga
*Data Science Department*
Eurecom, France

***Abstract*—Deep Neural Networks (DNNs) process big datasets achieving high accuracy on incredibly complex tasks. However, this progress has led to a scalability impasse, as DNNs require massive amounts of processing power and local memory to be trained, making them impossible or impractical to be used on a single device. This situation has led to the design of distributed training architectures, where the DNN and the training data can be split among multiple processors. How to choose the appropriate distributed training architecture, however, remains an open question. To help bring insights into this debate, in this work we design a Distributed Training Simulator (DTS) that estimates the training time of a DNN in a distributed architecture through a mathematical model of the distributed architecture and resource-allocation heuristics. We illustrate the power of the proposed DTS through the implementation of five different distributed architectures, Pipeline Learning, Federated Learning, Split Learning, Parallel Split Learning, and Federated Split Learning, and we validate the accuracy of the training estimates using three different datasets of varying complexity and two different DNNs. Finally, we present a trade-off analysis to demonstrate the coherence of DTS estimates for diverse high-performance computing scenarios by comparing these estimates with the behaviors of a real computer cluster.**

***Index Terms*—big data, distributed computing, deep learning, simulator.**

## I. Introduction

The interest in Deep Neural Networks (DNNs) has grown in the past decade, due to their ability to reach high accuracy levels when predicting outcomes for a variety of applications. A few examples of these applications are sensor fusion [1], malware detection [2], face recognition [3], robotics [4], and healthcare [5]. Training such complex models in a centralized fashion is becoming unsustainable, based on the ever-growing need to retrain a model with new, fresh data in a timely fashion. Training a neural network is costly both in terms of processing power and required memory. Additionally, there is a growing concern about how deep learning applications might invade privacy [6], [7]. These factors have led to the design of distributed training architectures [8]–[12].

Researchers have recently proposed methods and solutions aimed at increasing the accuracy of DNNs or decreasing DNNs' training time by focusing on several different characteristics of a DNN ecosystem. For example, some researchers have proposed tuning the hyper-parameters of a DNN [13], [14] or modifying its training sequence [15]. Some authors have proposed mathematical models for the general workload of DNNs [16], but have not expanded these models to distributed architectures and split computing. Others have focused on intra-DNN batch management optimizations [17], for different cores and threads within the same machine. Some of these studies have also focused on Split Computing (SC) applications [18]–[20], albeit only with a single client and a single server. However, none of them have focused on trying to optimize the very first choice that must be made when performing distributed training: selecting the appropriate neural network architecture, as well as the hardware configuration. This choice should allow the distributed training of the models to comply with the user's available resources while optimizing training time, or while keeping in mind the training time deadline to avoid service level agreement violations.

An immediate benefit of this optimized selection is minimizing the training time for the user's DNN. This minimal training time allows the user to retrain more often within a limited period and with fewer resources, leading to lower hardware costs, potentially higher revenues [21], and a decreased carbon footprint [22], [23].

Hence, there exists a gap in the study of Deep Learning optimization: to our knowledge, there is no work focused on the optimization of multi-client and multi-server training of DNNs. Additionally, none of these works provide mathematical models for the time it would take for a DNN to be trained on a distributed architecture with multiple clients and servers, nor do they use similar models to optimize the selection of the appropriate hardware and network configuration (network graph) for this architecture.

In this paper we aim to close this gap by presenting two key contributions: (1) We propose a simulator that, given information about the DNN and the cluster it will be trained on, outputs the final training time of the DNN on five distributed architectures: Pipeline Learning (PL) [8], Federated Learning (FL) [9], Split Learning (SL) [10], Parallel Split Learning (PSL) [11], and Federated Split Learning (FSL) [12]. Our Distributed Training Simulator (DTS) also outputs the network graph specifying the role (server, edge server, or client) each processor in the cluster or machine network must have to obtain the simulated output. (2) We provide the mathematical models used by the simulator to estimate the training time per batch of a DNN working on each of these five distributed architectures. The simulator obtains close-to-optimal solutions for these mathematical models through resource-allocation heuristics.

2375-0227/22/$31.00 ©2022 IEEE
DOI 10.1109/MASCOTS56607.2022.00011

The rest of the paper is organized as follows. Section II presents how our proposed simulator works, what information it requires from the user and its outputs. Section III details how the five distributed architectures work and how their workflow leads to our proposed mathematical models. Section IV details the experiments to validate the accuracy and coherence of DTS. Finally, in Section V we conclude by summarizing the most important insights from this research.

## II. Distributed Training Simulator (DTS)

This section explains the main workflow of the proposed Distributed Training Simulator (DTS) (Section II-A), what user-input information it requires (Section II-B), how it uses this information (Section II-C), its outputs (Section II-D), and its accompanying measurement tools (Section II-E).

### A. DTS Workflow

DTS' workflow is as follows: given the properties of a DNN (*what* will be trained), the available machines in a cluster (*where* the DNN will be trained), and information about the training sequence (*how* it should be trained), DTS outputs the lowest possible training time of that DNN for each of the supported distributed architectures and their required network graph. The network graph refers to how the machines should be linked through the network and which role (client or server) each machine should have in the distributed architecture. A graphical representation of DTS is shown in Figure 1.

### B. DTS Inputs

The user must provide three inputs: (1) *Cluster information*, telling DTS the characteristics of the HPC cluster or general machine network where the DNN will be trained, (2) *DNN information*, specifying the characteristics of the DNN to be trained, and (3) *Training sequence information*, specifying the setup of the training sequence that DTS must consider when calculating the final training time of the DNN. A list of these three groups of variables is shown in Table I.

**Cluster Information**: This set provides DTS with an accurate depiction of the machines in the network that can be used for DNN training and how they are interconnected. This set includes the number of processors in the network, their processing power, the potential fluctuation, their memory, the links between processors, their respective bandwidths, and how these can fluctuate depending on the network conditions.

**DNN Information.** This set provides information that allows DTS to allocate processing, memory, and communication resources according to what is required by the DNN. It includes the number of layers, the DNN's size in memory, its computational complexity (FLOPs), the size of gradients when being sent through the network, and others.

**Training Sequence Information.** This set gives DTS the constraints imposed by the desired training sequence for a DNN. It includes the sample size, the batch size, the number of batches per client, the total epochs, and the number of clients.

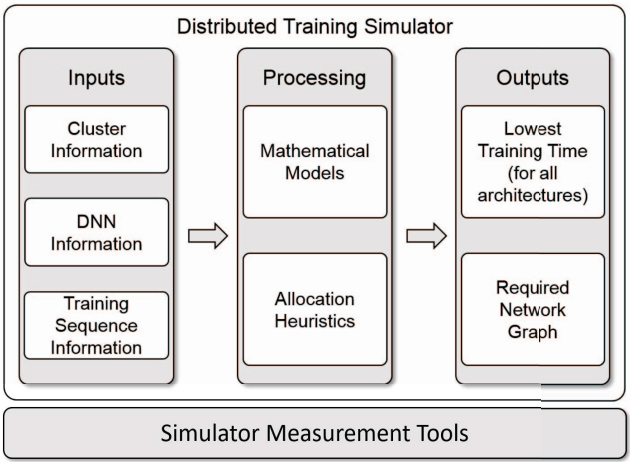



Fig. 1: Simplified graphical representation of the Distributed Training Simulator (DTS)'s workflow.

TABLE I: DTS's customizable parameters. All parameters are independent, which allows the user to simulate different DNNs on multiple machines and network conditions.

| Group | Variable | Unit |
|---|---|---|
| Cluster Information | Processor amount | None |
| | Processing power | GFLOP/s |
| | Power fluctuation | ±% |
| | Residual memory | GB |
| | Bandwidth | GB/s |
| | Bandwidth fluctuation | ±% |
| | Parallelization factor | None |
| DNN Information | Layer amount | None |
| | DNN complexity | GFLOP/sample |
| | Memory required | GB |
| | Server-side complexity | GFLOP/sample |
| | Client-side complexity | GFLOP/sample |
| | Server memory required | GB |
| | Client memory required | GB |
| | Size of weights | GB |
| | Size of gradients | GB |
| | Intermediate results | GB |
| Training Sequence Information | Individual sample size | GB |
| | Batch size | None |
| | Batches per client | None |
| | Training epochs | None |
| | Number of clients | None |

### C. DTS Processing

DTS processes the input to obtain the training time of the DNN in five distributed training architectures: PL, FL, SL, PSL, and FSL. To do so, the simulator uses allocation heuristics to solve mathematical models for the training time per batch of data of a DNN in each of these architectures.

**Mathematical models.** These models provide DTS with a mathematical representation of the training time per batch of data for a DNN in a distributed training architecture. The models account for the time to transfer data between machines, the time to perform forward and backward passes, and others. The models for the five currently supported architectures are described in detail in Section III.

**Allocation heuristics.** DTS's goal is to minimize the mathematical models while abiding by the constraints established by the input parameters in Table I. This requires assigning "roles" to processors in the cluster. Once the roles are defined, the characteristics of each chosen processor (processing power, memory, links to other processors, etc.) are introduced into the mathematical models to obtain a training time for the DNN in the corresponding distributed architecture. The assignable roles to the processors depend on the distributed architecture itself. Section III explains each of these roles in more detail.

The selection of these processors and their respective roles is not done randomly. DTS uses a unique resource-allocation heuristic for each distributed training architecture, but they all follow three key "guidelines": (1) the most CPU/GPU intensive tasks should be assigned to the most powerful processors, (2) no processor should execute a set of tasks that exceeds its available memory, and (3) the processors with the best links to others (highest bandwidths) should be assigned roles with the highest network overhead.

A summarized version of our heuristic works as follows. Firstly, DTS "ranks" the processors based on the availability of three resources: processing power, memory, and bandwidth. DTS then builds the specified distributed architecture by assigning roles to the highest ranked processors in each category. After each role is assigned, available resources are updated and the processors are re-ranked before the next assignation.

After each selection, DTS checks if all the necessary constraints have been met. If not, then DTS will determine which processor is causing the greatest "delay" in the training sequence, and either change its role or choose the next processor best suited for that task. This process continues until a set of processors that meets all constraints has been found.

Each of these successful selections is called a "path" that can lead to the optimized distributed training architecture. However, multiple paths can exist for any specific allocation scenario. As such, the user defines a maximum number of paths that DTS can obtain through the use of the *max_paths* parameter. If *max_paths* is set to infinite, DTS will attempt all possible configurations and reach the optimal selection. Once the maximum number of attempts has been reached, DTS will output the path that resulted in the lowest training time.

### D. DTS Outputs

DTS provides two outputs per distributed architecture: the lowest training time and the network graph.

**Lowest training time.** After having run all allocation heuristics, DTS outputs the lowest training time of the DNN for the supported distributed architectures. The user can then compare the training time of their DNN between each of the distributed architectures and make a decision based on their current goals, available resources, and time restrictions.

**Network graph.** All the output training times will be accompanied by a corresponding network graph that tells the user which role each processor in the network must have to achieve the proposed lowest training time. More information on each "role", how these architectures work, and their respective mathematical models are presented in Section III.

### E. DTS Measurement Tools

Many of the input parameters mentioned in Section II-B are difficult to obtain through specifications sheets or from PyTorch [24] code. For example, the computational complexity (total GFLOPs) of a DNN in PyTorch is not specified, and the processing power allocated by the operating system (OS) to the training of a DNN is not constant. Hence, we provide two measurement tools, one for the *DNN information* and one for the *processing power*, to help the user obtain these values.

The first tool obtains all *DNN information* from a DNN class written in PyTorch and then transforms it into usable parameters by the simulator. As an example, to obtain the computational complexity (GFLOPs) of the DNN, this tool uses the PyTorch computational complexity estimator [25] to estimate the total multiply-accumulate operations (MACs) needed to run a forward pass of one input sample. These MACs are then converted into GFLOPs. Other similar operations are done to obtain the remaining DNN-related parameters for DTS.

The tool for the *processing power* determines the total processing power a machine in the cluster allocates to the training task. The specification sheet of a conventional machine will detail the peak processing power, but the OS rarely allocates all resources to a single task. To calculate this parameter, the tool runs a short training session of the DNN using a small number of batches. The tool knows how many GFLOPs are required per sample, as obtained by using the first tool. Hence, the tool knows how many batches are used during the training session, how many GFLOPs are executed, and how long it takes for those operations to finish. With these three values, the tool calculates the processing power allocated by the OS in GFLOP/s. We remind the reader that in some HPC clusters it is possible to manually set the processing power allocated to a specific training task. If the user has access to such a tool, they can simply input this information directly into DTS without the use of our processing power estimation tool.

## III. Deep Learning Distributed Training Architectures and Mathematical Models

### A. Pipeline Learning (PL)

Pipeline Learning (PL), also referred to as Pipelining, was designed to address the need to train massive DNNs that require too much local memory to be housed in a single processor [8]. This distributed training architecture aims to split the NN into multiple processors in a network, where each processor houses a layer or group of layers of the NN. All processors are assigned the role of "workers" in this architecture. The mathematical model for the training time of one batch of data for a DNN in this architecture is:

$$T_{PL} = \sum_{i=1}^{m} \sum_{j=1}^{n} [T_{fe_{jj'}} + T_{F_{ij}} + T_{B_{ij}} + T_{wr_{jj''}}], \qquad (1)$$

where $T_{PL}$ is the training time for one batch of a neural network (forward + backward pass), $T_{fe}$ is the time it takes for processor $j$ to fetch the data from a previous processor $j^{'}$, $T_{F_{ij}}$ is the time it takes processor $j$ to compute the forward pass of layer $i$, $T_{B_{ij}}$ is the time it takes processor $j$ to compute the backward pass of layer $i$, and $T_{wr_{jj''}}$ is the time it takes for a processor $j$ to write (output) the data to the next processor $j^{''}$, $m$ is the total of layers in the DNN, and $n$ is the number of processors necessary to house all layers of the DNN.

### B. Federated Learning (FL)

Federated Learning (FL) is an architecture designed to train machine learning models with sensitive data from multiple users at once, with the constraint that the data can never be shared between them. As such, it is a privacy-preserving, completely parallelized architecture [9].

The processors in FL can be assigned the roles of "clients" or "servers". Clients have access to the data and train the entire DNN on their processors, but they cannot share the data between them. Servers are only in charge of a process called "federated averaging", where the server takes all trained models from the clients and averages their weights to obtain a new, universal DNN model. This model is then sent back to the clients for the start of the next epoch of training.

Hence, the mathematical model for the training time of one batch of data for any DNN in this distributed architecture is:

$$T_{FL} = \max[T_{F_j} + T_{B_j}] + \max[2 * T_{tr_{jk}}] + T_{avg_k} \qquad (2)$$

where $T_{FL}$ is the training time of the DNN for one batch of data, $T_{F_j}$ is the time it takes for client $j$ to compute the forward pass of the DNN for one batch, $T_{B_j}$ is the time it takes for client $j$ to compute the backward pass of the DNN for one batch, $2 * T_{tr_{jk}}$ is the time it takes to transfer the weights through the network from the $j$-th client to the $k$-th server, and $T_{avg_k}$ is the time it took the server to average the models. $T_{tr_{jk}}$ is multiplied by 2 because the total training time for one batch includes sending the weights to the server and then receiving them back after the federated averaging process.

### C. Split Learning (SL)

Split Learning (SL) was designed to provide advantages of both PL and FL: the need to split a DNN into multiple processors (PL), while also training on multiple datasets from different clients without sharing the data between them [10]. SL splits a DNN into two parts: server-side and client-side. The server-side split is kept on a central server, while the client-side split is sent to all individual clients in the network.

As with FL, the processors can be assigned roles as either clients or servers. Clients have access to the data and train the client-side DNN, while servers do not have access to the raw data and only train the server-side DNN. This is a sequential architecture because the training with one client must end before proceeding on to the training of the next client.

The mathematical model for the training time of one batch for a DNN in the SL architecture is:

$$T_{SL} = \sum_{j=1}^{n}[T_{fe_j} + T_{F_j} + T_{B_j} + 2 * T_{tr_k} + T_{F_k} + T_{B_k}] + (n-1) * T_{trw_{j-j+1}} \qquad (3)$$

where $T_{SL}$ is the time it takes to train a DNN for one batch in SL, $T_{fe_j}$ is the time it takes for client $j$ to fetch its data, $T_{F_j}$ is the time it takes for client $j$ to compute the forward pass of its client-side NN, $T_{B_j}$ is the time it takes for client $j$ to compute the backward pass of its client-side NN, $2 * T_{tr_k}$ is the time it takes to transfer data from the client to server $k$, $T_{F_k}$ is the time it takes the server to process the forward pass of the server-side NN, $T_{B_k}$ is the time it takes the server to process the backward pass of the server-side NN, and $n$ is the number of clients. Notice that $T_{tr_k}$ is multiplied by 2 because each client sends data to the server once during the forward pass (intermediate results) and once during the backward pass (gradients). Finally, $T_{trw_{j-j+1}}$ is the time it takes for the weights to be transferred from client $j$ to client $j+1$ (the next one in the chain), which is done $(n-1)$ times.

### D. Parallel Split Learning (PSL)

Parallel Split Learning (PSL) combines the benefits of splitting a DNN (PL, SL) with the parallelization strategies of its client side (FL) [11]. It is very similar to both SL and FL in the fact that it has multiple clients and a single central server. As such, processors can be assigned the role of either clients or servers. However, its client-side training phase is parallelized, while the server-side remains sequential.

The mathematical model for the training time of one batch of data for a DNN in PSL is:

$$T_{PSL} = \max[T_{fe_j} + T_{F_j} + T_{B_j} + 2 * T_{tr_j}] + \sum_{j=1}^{n}[T_{F_j} + T_{B_j}] \qquad (4)$$

where $T_{PSL}$ is the time to train a DNN for one batch, $T_{fe_j}$ is the time it takes for client $j$ to fetch its data, $T_{F_j}$ is the time it takes client $j$ to compute the forward pass of its client-side DNN, $T_{B_j}$ is the time it takes client $j$ to compute the backward pass of its client-side DNN, $2 * T_{tr_j}$ is the time it takes for client $j$ to transfer its data to the server (send and receive), $\sum_{j=1}^{n}[T_{F_j} + T_{B_j}]$ is the time it takes the server to process the forward and backward passes of the server-side DNN for all $j$ clients and $n$ represents the total number of clients.

### E. Federated Split Learning (FSL)

FSL aims to combine all the benefits of the previous architectures and add a focus on privacy-preserving measures [12]. To do so, in addition to both clients and servers (also called "parameter servers"), FSL introduces a new processor role called "edge server". Clients have access to the data and the client-side DNN, and each client in the network has an accompanying edge server which helps the client train the DNN by splitting it at least once. As in FL, servers are only in charge of federated averaging.

The mathematical model for the training time of one batch for a DNN trained with FSL is:

$$T_{FSL} = \max[T_{fe_j} + T_{F_j} + T_{B_j} + 2 * T_{tr_{jj'}} + T_{F_{j'}} + T_{B_{j'}} + T_{tr_{j'j''}}] + T_{avg} \quad (5)$$

where $T_{FSL}$ is the time it takes for a DNN to be trained for one batch in FSL, $T_{fe_j}$ is the time it takes client $j$ to fetch its data, $T_{F_j}$ is the time it takes client $j$ to process its forward pass, $T_{B_j}$ is the time it takes client $j$ to process its backward pass, $2 * T_{tr_{jj'}}$ is the time it takes client $j$ to transfer data to edge server $j'$ (send and receive), $T_{F_{j'}}$ is the time it takes for the edge server $j'$ to compute its forward pass, $T_{B_{j'}}$ is the time it takes for the edge server $j'$ to compute its backward pass, $T_{tr_{j'j''}}$ is the time it takes the edge server $j'$ to transfer the data to the server $j''$, and $T_{avg}$ is the time it takes the server to perform a federated average of all the received models.

### *F. Additional Distributed Training Architectures*

The current version of DTS supports five distributed training architectures, but more can be added. To do so, the user must provide a resource-allocation heuristic and a mathematical model to estimate the training time of the new architecture. Any user can write these two elements in a Jupyter Notebook [26] file and then instruct DTS to use this code. To facilitate this process, the user has access to all the global variables used by the current allocation heuristics, so that the integration of the new model is as seamless as possible.

## IV. Evaluation

This section details the experiments to validate the *accuracy* and *coherence* of DTS when estimating the training time of a DNN. Section IV-A details our hardware and software configuration, the datasets, and the DNNs used for our testbed. Sections IV-B and IV-C detail the two main accuracy validation experiments we conducted. Section IV-D details the tradeoff analysis we performed to validate the coherence of all simulations, i.e. how close the behavior of the simulated architectures was to the behavior of their real-life counterparts.

### *A. Experimental Setup*

**Configuration:** All experiments were carried out by creating custom versions of VGG-11 [27], because of its widespread use and ease of reproducibility. These versions are referred to as DNN 1 and DNN 2, shown in Tables II and III respectively. They were trained on a computer with a CPU Intel Core i7-8550U @ 1.80GHz with 8 GB of RAM, running on Ubuntu 20.04.2 LTS. We then used PySyft [28] to implement these DNNs in PL, FL, SL, PSL, and FSL.

**Datasets:** Three datasets of varying complexity were used for these experiments. Firstly, MNIST [29] was used as the most basic dataset, which consists of 1000 images of the handwritten numbers 1 through 9. Secondly, Cats and Dogs [30], a popular dataset of 25000 images, was used; we considered this dataset to be one of moderated complexity. Finally, the EchoNet-Dynamic dataset [31] was used as the dataset with a high degree of complexity. Echonet-Dynamic contains over 10000 echocardiography open-source videos. The purpose of this dataset is to challenge the research community to build AI models to diagnose cardiovascular diseases and abnormalities.

TABLE II: DNN 1, detailed in PyTorch nomenclature. It has 72802 parameters (weights). The Rectified Linear Unit (ReLU) activation function is not listed for brevity.

| Layer Number | Layer Details |
|---|---|
| 1 | Conv2d(1, 16, kernel_size=(5, 5)) |
| 2 | maxpool (input, (2,2)) |
| 3 | Conv2d(16, 32, kernel_size=(5, 5)) |
| 4 | maxpool (input, (2,2)) |
| 5 | Conv2d(32, 64, kernel_size=(5, 5)) |
| 6 | maxpool (input, (2,2)) |
| 7 | Linear(in_features=256, out_features=32) |
| 8 | Linear(in_features=32, out_features=2) |
| 9 | soft-max |

TABLE III: DNN 2, detailed in PyTorch nomenclature. It has 1044482 parameters (14 times the amount of DNN 1) and is 4.8 times more computationally expensive. The Rectified Linear Unit (ReLU) activation function is not listed for brevity.

| Layer Number | Layer Details |
|---|---|
| 1 | Conv2d(1, 32, kernel_size=(3, 3)) |
| 2 | maxpool (input, (2,2)) |
| 3 | Conv2d(32, 64, kernel_size=(3, 3)) |
| 4 | maxpool (input, (2,2)) |
| 5 | Conv2d(64, 128, kernel_size=(3, 3)) |
| 6 | maxpool (input, (2,2)) |
| 7 | Conv2d(128, 256, kernel_size=(3, 3))) |
| 8 | maxpool (input, (2,2)) |
| 9 | Linear(in_features=1024, out_features=512) |
| 10 | Linear(in_features=512, out_features=256) |
| 11 | Linear(in_features=256, out_features=2) |
| 12 | soft-max |

For the experiments in Sections IV-B and IV-C, the datasets were distributed amongst the processors as follows. Firstly, all architectures had at least three clients. Depending on the architecture, the number of servers would be chosen appropriately from the other virtual workers. Regarding the data, PL required the entire dataset to be stored on a single virtual worker, as the architecture does not allow for split data. For FL, the training data (including labels) was split into three, one for each virtual client. For SL and PSL, the training data was split into three, one for each client, but the labels were stored in the architecture's central server. Finally, for FSL, the training data was split into three, one for each client, and the labels were stored in the edge servers paired up with each client. Regarding the split of the DNNs, DNN 1 (Table II) had layers 1-4 stored in each client, while layers 5-9 were stored in the servers. For DNN 2 (Table III), layers 1-3 were stored in the clients, while layers 4-12 were stored in the servers.

### *B. Accuracy Validation Metric #1: Batch Training Time Estimation (BTTE)*

These experiments measure DTS's accuracy for estimating the time required to train with a single batch. The

Batch Training Time Estimation (BTTE) experiments were executed as follows: (1) For each DNN-dataset-architecture configuration, five epochs of training were executed in our testbed, where each epoch contained at least 100 batches. This leads to a total of 500 batch testbed times for each DNN-dataset-architecture configuration. (2) We simulated 100 batch times for each DNN-dataset-architecture configuration. (3) We compared these 100 simulated batch times to 100 testbed batch times sampled from the 500 testbed times. This sampling process was done to account for potential differences in batch times between epochs: the 100 testbed batch times were randomly uniformly sampled as 20 batches from each epoch. Based on the comparison results, the average accuracies for each architecture among the three datasets were calculated (Table IV). To better visualize the significance of these accuracies, we also plotted the simulated and testbed batch times against each other in boxplots (Figure 2).

As a reminder, in our testbed, many of the hardware characteristics are completely controlled and allocated over time by the OS. We used the measurement tools to estimate these properties and DTS's modifiable parameters to simulate this pseudo-stochastic behavior. Despite all these approximations, DTS still managed to achieve an accuracy of over 80% in all BTTE scenarios. In an HPC cluster, the accuracy of DTS would increase, as the user would have more control over the allocation of these resources over time. This high accuracy validates DTS's output for batch times. Hence, the next step is to validate our prediction accuracy of entire training epochs.

### *C. Accuracy Validation Metric #2: Epoch Training Time Estimation (ETTE)*

Epoch Training Time Estimation (ETTE) is very similar to BTTE, except that this time we measure and compare the times for entire epochs, not individual batches. Each epoch consisted of training the DNN with the totality of the data: over 10000 images for MNIST, over 25000 images for Cats and Dogs, and over 37000 images for Echonet-Dynamic, divided into multiple batches, each containing 100 images.

As before, we compare our testbed epoch times with our simulated epoch times and calculated the accuracies for each architecture's predictions (Table IV). Better visualization of the significance of these accuracies is shown in Figures 2c and 2d. Considering the estimation error of a predicted batch may accumulate over time when predicting epochs, it is natural for the accuracies in this experiment to be either very similar to or lower than the BTTE results. The biggest reductions in accuracy were only 7.59% for PSL for DNN 1 and 7.66% for PL for DNN 2. Considering everything is being simulated solely through our resource-allocation heuristics, mathematical models, and measurement tool approximations, we consider these small reductions in accuracy as acceptable results.

### *D. Coherence Validation: A Tradeoff Analysis*

In this section, we extrapolate DTS's results to more resource-intensive DNN training sessions, which are typical in HPC scenarios. We validate the coherence of DTS's outputs by performing a tradeoff analysis. To understand the methodology of this set of experiments, it is important to remember that DTS allows for the independent manipulation of all variables in a scenario. This gives the user the freedom of changing any single simulation parameter, running DTS, and then analyzing the impact of said parameter on the obtained results. The HPC cluster properties for these experiments were taken from real-life Amazon Web Services (AWS) clusters [32]. We simulated over 36000 scenarios, with different DNN sizes, cluster properties, and network conditions. The results were analyzed to check if they were coherent with how these distributed architectures would behave in a real-life HPC cluster.

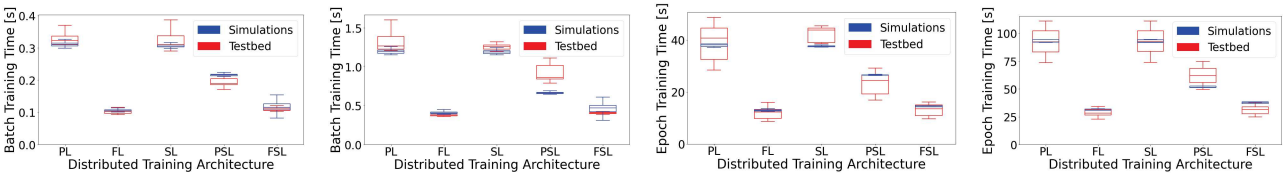
TABLE IV: Summary of results for BTTE and ETTE. The lowest accuracy was reported for PSL on ETTE (73.61%), while the highest accuracy was reported for SL on BTTE (95.70%). Considering all architectures and both validation metrics, this amounts to an average accuracy of 84%.

| DNN | Architecture | BTTE Acc. | ETTE Acc. |
|---|---|---|---|
| DNN 1 | PL | 0.8389 | 0.8019 |
| | FL | 0.8178 | 0.7984 |
| | SL | 0.8950 | 0.9139 |
| | PSL | 0.8120 | 0.7361 |
| | FSL | 0.8136 | 0.7750 |
| DNN 2 | PL | 0.9477 | 0.8711 |
| | FL | 0.8986 | 0.8308 |
| | SL | 0.9570 | 0.8953 |
| | PSL | 0.8024 | 0.7969 |
| | FSL | 0.8246 | 0.7831 |

To this aim, we plotted the training time vs. one parameter with a fixed value, while another parameter randomly changed its value from simulation to simulation. Simulation results were obtained with a 95% confidence interval with respect to the randomly changing parameter. This process was repeated for all potential combinations of DTS's parameters including processing power, power and bandwidth fluctuation, residual memory per node, memory required to run the neural network, and others that we will release with the git repository of this project upon acceptance. This procedure enabled us to determine how much effect the variability of the randomly changing parameter had on the training time of a DNN when the condition defined by the fixed parameter was met. Then we checked if these results were coherent with the expected behavior of these distributed architectures in an HPC cluster.

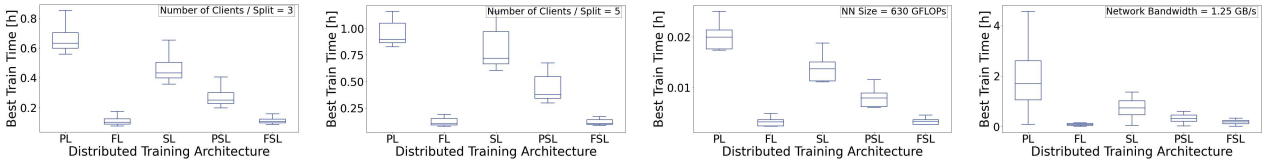
We first observe that PL, SL, and PSL are not scalable. As the number of clients increases, the difference between the parallelized architectures (FL, FSL) and the sequential architectures (PL, SL, PSL) also increases. This behavior is evident in DTS's outputs, as shown when comparing Figures 3a and 3b. These figures also evidence the fact that the worst results of the parallelized architectures are on par or even better than the best results of the sequential architectures.

Secondly, when the bandwidth is *high*, then the most influential variable should be the available processing power, assuming the DNN properties (size, number of layers, etc.) are fixed. In our simulated clusters *high* refers to a bandwidth $B \geq 40$ Gb/s. This behavior is present in DTS's outputs: it does




(a) (b) (c) (d)

Fig. 2: BTTE and ETTE results for all three datasets for DNN 1 and DNN 2. (a) Results for BTTE for DNN 1 with Echonet-Dynamic. (b) Results for BTTE for DNN 2 with MNIST, (c) Results for ETTE for DNN 1 for Echonet-Dynamic. (d) Results for ETTE for DNN 2 for Cats and Dogs.




(a) (b) (c) (d)

Fig. 3: Obtained best training times on different DTS configuration scenarios. (a) Variable processing power, fixed bandwidth of 40 Gb/s (5 GB/s), and 3 clients. (b) Variable processing power, fixed bandwidth of 40 Gb/s (5 GB/s), and 5 clients. (c) Variable processing power, fixed bandwidth of 40 Gb/s (5 GB/s), and network size of 630 FLOPs. (d) Variable network size, fixed bandwidth of 10 Gb/s (1.25 GB/s), and fixed processing power of 13K GFLOPS /s.

not matter if the DNN is small or big, the available processing power always causes the biggest variance in the final training time. This behavior is shown in Figure 3c, and it appeared in all scenarios where the bandwidth B $\geq$ 40 Gb/s (5 GB/s).

However, when the bandwidth is *low* then the bandwidth becomes the main bottleneck. In our case, *low* refers to a bandwidth B $\leq$ 10 Gb/s or lower. Now, most of the training time is spent sending information through the network instead of processing information on the clients and servers. As such, a parallelized architecture with a very high network overhead like FSL (because of its $2N+1$ processor requirement) should be much slower and even comparable to other sequential architectures with a much lower network overhead. This behavior is evident in Figure 3d, where FSL is now on par with a semi-sequential architecture like PSL. A high network overhead combined with low bandwidth and gigabytes of data being constantly sent over the network slowed down FSL to a point where its parallelization strategies showed no benefits.

Other behaviors were also present in this tradeoff analysis. For example, DTS would notify the user when there was not enough local memory available in the processors to house the DNN. The way a DNN is split also defined whether a distributed training architecture was viable or not. For example, splitting a DNN such that 50% of it is on the servers and 50% of it is on the clients usually led to scenarios where the training was not possible because the clients could not house that portion of the DNN. However, splitting a DNN 70%-30% would make the training scenario viable again, without the need to modify any properties of the HPC cluster.

## V. Conclusion and Discussion

In this work, we presented our Distributed Training Simulator (DTS), which given information about a DNN and the machine network it will be trained on, can output the best training time and the required network graph for five distributed training architectures: PL, FL, SL, PSL, and FSL. We accounted for fluctuations in processing power and bandwidth, datasets of different complexities, DNNs of vastly different sizes, and times from multiple batches and epochs in different points of the DNN training sequence. We presented the experiments to validate its training time estimation accuracy and the coherence of its results with real-life scenarios.

The most important takeaway is that our simulator can adequately output the time it takes to train a DNN for a specified number of batches and epochs in the five studied distributed training architectures with an average accuracy of 84.05% (minimum 73.51%, maximum 95.70%). Other concluding remarks include: (1) The difference between parallelized and sequential architectures is proportional to the number of clients. (2) Modifying the available processing power always drastically influenced the final training time of a DNN. (3) Low bandwidth can easily become a severe bottleneck when working with architectures with high network overhead like Federated Split Learning architectures.

Additionally, DTS can be extended to account for other DNNs and distributed training architectures. Regarding the DNNs, the provided measurement tools work for all kinds of DNNs input as a PyTorch class, which allows the user to input the DNN parameters into DTS. Regarding the distributed architectures, the mathematical models and allocation heuristics for the currently supported architectures are provided as a

base for future optimization strategies to be included in DTS. As a relevant example, the current version of DTS accounts for the pipelining architecture defined in [8], but other works have focused on specifically optimizing this strategy to its full potential [33]–[35]. While these approaches have not been included in this version of DTS, their integration into the simulator is straightforward and makes part of the future work.

## ACKNOWLEDGEMENTS

This work was supported by NSF award # 1908574 and by scholarship # 270-2019-238 from the Panamanian Institute for the Training and Development of Human Resources (IFARHU) and the Panamanian National Secretariat of Science, Technology, and Innovation (SENACYT).

## REFERENCES

[1] V. Radu, N. D. Lane, S. Bhattacharya, C. Mascolo, M. K. Marina, and F. Kawsar, "Towards multimodal deep learning for activity recognition on mobile devices," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, ser. UbiComp '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 185–188. [Online]. Available: https://doi.org/10.1145/2968219.2971461

[2] R. Feng, S. Chen, X. Xie, G. Meng, S.-W. Lin, and Y. Liu, "A performance-sensitive malware detection system using deep learning on mobile devices," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1563–1578, 2021.

[3] C. N. Duong, K. G. Quach, I. Jalata, N. Le, and K. Luu, "Mobiface: A lightweight deep learning face recognition on mobile devices," in *2019 IEEE 10th International Conference on Biometrics Theory, Applications and Systems (BTAS)*, 2019, pp. 1–6.

[4] J. Shabbir and T. Anwer, "A survey of deep learning techniques for mobile robot applications," 2018.

[5] Y. Deng, "Deep learning on mobile devices: a review," in *Mobile Multimedia/Image Processing, Security, and Applications 2019*, S. S. Agaian, V. K. Asari, and S. P. DelMarco, Eds., vol. 10993, International Society for Optics and Photonics. SPIE, 2019, pp. 52 – 66. [Online]. Available: https://doi.org/10.1117/12.2518469

[6] K. Degirmenci, "Mobile users' information privacy concerns and the role of app permission requests," *International Journal of Information Management*, vol. 50, pp. 261–272, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0268401218307965

[7] F. Mireshghallah, M. Taram, P. Vepakomma, A. Singh, R. Raskar, and H. Esmaeilzadeh, "Privacy in deep learning: A survey," 2020.

[8] T. Ben-nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys*, vol. 52, pp. 1–43, 2019.

[9] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. Aguera y Arcas, "Communication-efficient learning of deep networks from decentralized data," *International Conference on Artificial Intelligence and Statistics (AISTATS)*, vol. 54, 2017. [Online]. Available: https://arxiv.org/pdf/1602.05629.pdf

[10] O. Guptaa and R. Raskara, "Distributed learning of deep neural network over multiple agents," 2018.

[11] J. Jeon and J. Kim, "Privacy-sensitive parallel split learning," *IEEE*, 2020. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=9016486

[12] V. Turina, Z. Zhang, F. Esposito, and I. Matta, "Federated or split? a performance and privacy analysis of hybrid split and federated learning architectures," in *In proc. of IEEE Int. Conference on Cloud Computing (IEEE CLOUD 2021)*, online, sept 2021.

[13] Y. Dokuz and Z. Tufekci, "Mini-batch sample selection strategies for deep learning based speech recognition," *Applied Acoustics*, vol. 171, p. 107573, 2021.

[14] T. Yu and H. Zhu, "Hyper-parameter optimization: A review of algorithms and applications," 2020. [Online]. Available: https://arxiv.org/pdf/2003.05689.pdf

[15] S. Mittal and S. Umesh, "A survey on hardware accelerators and optimization techniques for rnns," *Journal of Systems Architecture*, vol. 112, p. 101839, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762120301314

[16] H. Qi, E. R. Sparks, and A. S. Talwalkar, "Paleo: A performance model for deep neural networks," in *ICLR*, 2017.

[17] S. L. Xi, Y. Yao, K. Bhardwaj, P. N. Whatmough, G. Wei, and D. Brooks, "SMAUG: end-to-end full-stack simulation infrastructure for deep learning workloads," *CoRR*, vol. abs/1912.04481, 2019. [Online]. Available: http://arxiv.org/abs/1912.04481

[18] H. Li, K. Ota, and M. Dong, "Learning iot in edge: Deep learning for the internet of things with edge computing," *IEEE Network*, vol. 32, no. 1, pp. 96–101, 2018.

[19] D. Jahier Pagliari, R. Chiaro, E. Macii, and M. Poncino, "Crime: Input-dependent collaborative inference for recurrent neural networks," *IEEE Transactions on Computers*, pp. 1–1, 2020.

[20] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," 04 2017, pp. 615–629.

[21] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 561–577.

[22] H. Alfauri and F. Esposito, "A distributed consensus protocol for sustainable federated learning," in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, 2021, pp. 161–165.

[23] L. F. W. Anthony, B. Kanding, and R. Selvan, "Carbontracker: Tracking and predicting the carbon footprint of training deep learning models," 2020.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[25] V. Sovrasov, "Flops counter for convolutional networks in pytorch framework," 2021. [Online]. Available: https://github.com/sovrasov/flops-counter.pytorch

[26] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87 – 90.

[27] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *ICLR*, 2015.

[28] T. Ryffel, A. Trask, M. Dahl, B. Wagner, J. Mancuso, D. Rueckert, and J. Passerat-Palmbach, "A generic framework for privacy preserving deep learning," 2018.

[29] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[30] M. Research, "Kaggle cats and dogs dataset," 2017. [Online]. Available: https://www.kaggle.com/c/dogs-vs-cats

[31] D. Ouyang, B. He, and e. a. Ghorbani, A., "Video-based ai for beat-to-beat assessment of cardiac function," *Nature*, vol. 580, p. 252/256, 2020. [Online]. Available: https://www.nature.com/articles/s41586-020-2145-8

[32] Amazon, "Recommended gpu instances," 2021. [Online]. Available: https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html

[33] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–15. [Online]. Available: https://doi.org/10.1145/3341301.3359646

[34] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *CoRR*, vol. abs/1811.06965, 2018. [Online]. Available: http://arxiv.org/abs/1811.06965

[35] J. Tarnawski, A. Phanishayee, N. R. Devanur, D. Mahajan, and F. N. Paravecino, "Efficient algorithms for device placement of DNN graph operators," *CoRR*, vol. abs/2006.16423, 2020. [Online]. Available: https://arxiv.org/abs/2006.16423