Recommender System for Optimal Distributed Deep Learning in Cloud Datacenters

Muhammad Hassaan Anwar, Saeid Ghafouri, Sukhpal Singh Gill and Joseph Doyle

School of Electronic Engineering and Computer Science, Queen Mary University of London, Mile End Rd, Bethnal Green, London E1 4NS, UK
m.anwar@se19.qmul.ac.uk, s.ghafouri@qmul.ac.uk, s.s.gill@qmul.ac.uk, j.doyle@qmul.ac.uk

***Abstract*— With the modern advancements in Deep Learning architectures, and abundant research consistently being put forward in areas such as computer vision, natural language processing and forecasting. Models are becoming complicated and datasets are growing exponentially in size demanding high performing and faster computing machines from researchers and engineers. TensorFlow provides a wide range of distributed deep learning high-level APIs to address this issue, that can scale deep learning training from one machine to more than one. In this paper, we have investigated the performance of computing clusters utilizing those APIs. We created clusters of different sizes and discuss performance issues of distributed deep learning under high latency and poor communication conditions. To address the challenge of finding the optimal cluster for fast distributed deep learning, we have proposed a recommendation system, that can provide an optimal cluster size for fastest training time, given batch size and networking latency. Our results show that using a 2 machine cluster is both faster and cheaper than a four machine cluster for certain algorithms when network delay is high.**

Keywords—Deep Learning, Cloud Computing, TensorFlow, AWS, Azure, Distributed Systems

I. INTRODUCTION

Deep Learning has been fueling the growth of state of the art Artificial Intelligence (AI) applications. Building more dense deep learning models also shows great results in classification accuracy (SIMONYAN AND ZISSERMAN, 2015). However, training models using a single machine could take a long time (Gill et al. 2019). As an example, the training time of the Resnet-50 model with a batch size of 256 images, on around one and a half million images takes approximately 29 hours on a single machine with 8 x Tesla P100 GPUs (Goyal et al., 2018). A solution was proposed by (NATU AND GHOSH, 2019), where they suggested the use of Distributed training for deep learning. Previously, (Feng, Xiang, and Zhou, 2016) demonstrated a 24x speedup with the help of distributed training (Moritz et al., 2016) showed an order of magnitude decrease in the training time of models utilizing a unified interface with the ability to perform actor based computations.

With the benefits of distributed training in optimizing training time, there arises an issue of scalability limits with the distributed training cluster (Keuper and Pfreundt, 2016). One of the reasons for these limitations could be high latency between machines (Veeramanikandan et al., 2020), as machines that are part of a cluster can be in a different physical location, causing communication delays. Therefore, scaling machines for a given batch size beyond a certain limit may have adverse effects on the training time, making the entire machine learning production environment inefficient (Gill et al. 2020).

Determining the optimal size of a cluster is very challenging as the processing time for a deep learning workload is dependent on both the networking conditions between the machines and the particular algorithm utilized. Mahajan *et al.* have recently examined multiple machine learning algorithms deployed in a Microsoft cluster and determined that some algorithms will perform comparably in a distributed deployment while others will suffer significant performance degradation (Mahajan et al. 2020*).* For example, they found that the performance of the VGG16 algorithm drops by 20% when distributed over two machines due to the very large number of parameters while the performance of the Inception-v3 algorithm is similar (Suresh et al., 2019a). Themis, however, is designed to obtain optimal performance in a GPU cluster while our work focuses on selecting the optimal cloud resources for a job (Suresh et al., 2019b). Cost is not considered in Themis as the hardware has been purchased and is always available. Our work examines using cloud resources in short time frames for optimal performance. In addition, other complications such as different computing architectures and legacy systems further complicate this problem. Unfortunately, there is not an exhaustive list of the performance of deep learning algorithms under various networking conditions. Thus, selecting the correct cluster is very difficult to achieve without experimentation or code analysis. This work serves as a step to provide a system which can recommend cluster sizes for deep learning algorithms based upon experimentation.

In this paper, we will discuss how utilizing an appropriate amount of computation nodes, researchers and engineers can take complete advantage of distributed training under un-favourable network environments for training their deep learning models under minimal time. Later in this paper, we propose a model that can provide an optimal cluster size for a given deep learning workload. Our model provides optimal cluster size prediction for faster training speed for a given deep learning workload. The key contributions of this paper are as follows:

- We analysed the performance of a machine learning algorithm to assess its performance on a variety of cluster sizes with different networking conditions and algorithm parameters on the Microsoft Azure and AWS cloud platforms.
- Using this analysis we designed and implemented a recommendation system to automatically select the most efficient cluster size based upon networking conditions and algorithm parameters.

The rest of the paper is structured as follows: Section 2 presented the preliminaries used in this work. Section 3 discusses the existing literature. Section 4 presents the problem formulation and methodology is given in Section 5. Section 6 presents the performance evaluation and Section 7 discuss the experimental results. Section 8 concludes the paper and highlights the future directions.

II. PRELIMINARIES

To understand distributed training, it is important to understand about Stochastic Gradient Descent (SGD) that serves as the underlying principle behind any Distributed Training method.

A. Distributed Training:

There are several ways for training machine learning models at scale. However, the adoption of deep learning requires a low level understanding of APIs provided by TensorFlow (TensorFlow Org., 2020) and MxNet (Chen et al., 2016). In this paper, we have used TensorFlow's Multi Worker Mirrored Strategy API, we implemented distributed training on MNIST Dataset(Keras Team, 2020) and discussed its performance on clusters comprising of multiple Azure and AWS (CPU only) virtual machines. We also discussed how latency and network limitations affect performance.

B. SGD: Stochastic Gradient Descent:

Gradient Descent is generally for optimizing non-convex objective functions, that attempts to minimize loss function using forward and backward propagation in an iterative way (Bottou, 2010). Training models on entire datasets using full batches can be time consuming. The author has also discussed how stochastic gradient descent shows excellent performance by utilizing subsets of data for training. SGD utilizes randomly selected mini-batch sizes B, providing stable convergence at fair computational costs on a single node (Keuper and Pfreundt, 2016). Batch SGD requires significantly lesser computation capacity compared to full gradient descent. Both algorithms have similar ambitions (with no guarantee) of converging to the global maximum. The algorithm converges by adjusting weights, as shown in the equation below:

$$\theta_{t+1} \leftarrow \theta_t - \epsilon \partial_\theta x_j(\theta_t) \qquad (1)$$

Here $\theta t + 1$ is the new state that is achieved by applying updates on θt (the previous state). Each update is made on a small set of training samples (mini-batch). (Keuper and Pfreundt, 2016) mentioned that scaling SGD is possible in two ways, either computing updates faster or increasing the size of steps (ϵ). They also mention that both methodologies are difficult to achieve in a distributed setting.

C. Distributed training:

Neural Networks rely on gradient descent updates for optimal parameters through back propagation (Tuli et al, 2021). To train Neural Networks in a parallel architecture, it is necessary to synchronize all the gradients to achieve a global state across devices, which will allow a worker to proceed to its next step (Xu et al, 2020). The gradient updates are required very frequently (for each mini-batch).

1) Model Parallel:

In this fashion, the model is distributed into multiple devices (machines/virtual machines), which can lead to several critical issues such as un-balanced model pre-partitioning, latency, and heavy dependency management (ZHANG ET AL., 2018). This framework is particularly useful in case, where models cannot be held within one device. Model placement can be a challenge for this framework. In this case, placing parts of neural models on devices could be learned by reinforcement learning that helps in optimizing device placement for computational graphs (BOTTOU, 2010)

2) Data Parallel:

In Data Parallel methods, the model is replicated in all devices (machines) and data is partitioned between all worker nodes. Workers train synchronously and aggregate gradient at each step to train the central model.

3) Parameter Server Strategy:

In a parameter server (asynchronous gradient descent) framework, each worker is initialized with a weight "w". The dataset is divided among all devices. A sub-gradient is calculated at each step using a worker's data and weight. The "parameter server" (or master) itself is a single machine that takes the responsibility of maintaining the most up to date parameters. The worker

sends a sub-gradient to the master (the parameter server), the master acknowledges the sub-gradient, updates the weight, and sends updated weight back to the worker. The master performs this step for each worker available, allowing all workers to finish this step asynchronously. However, if the sub-gradient from Worker B arrives at the master node when the master node is communicating with Worker A then the gradient update of worker B cannot be started until gradient update of worker A is finished. This indicates a locking mechanism that avoids weight update conflicts on the master machine.

Restricting the master machine to process one sub-gradient at a time. Hogwild method (Recht et al., 2011) can be considered a variant of Asynchronous SGD where a shared memory maintains the global state. It removes the lock and allows multiple sub-gradients to be processed by the master node at the same time.

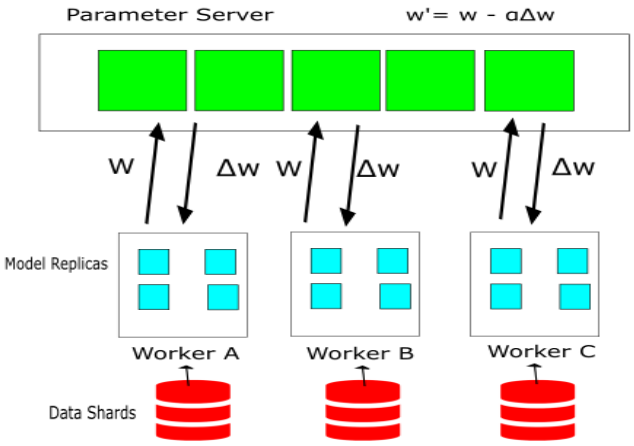

Fig. 1. A simple parameter server strategy demonstration

D. All-Reduce Algorithm:

TensorFlow uses the All-Reduce algorithm for its Multi Worker Mirrored strategy. All-reduce algorithm, all devices exchange information on local updates that they collect on each step. In Figure 1, there are 3 CPUs, each performing a single step that updates the mirrored version of three variables, each ball blue, green, and red corresponds to a variable. For a single step different updates are made on different devices and once the step is performed on all devices, it gets propagated in a circular fashion between other devices. At that point, all devices will have all updates from all other variables requiring n-1 transfers for n devices. When all updates have been collected, a reduce function can be used to combine the updates to a single global value. Common reduce functions are either sum or average of those updates as shown in Fig. 2.

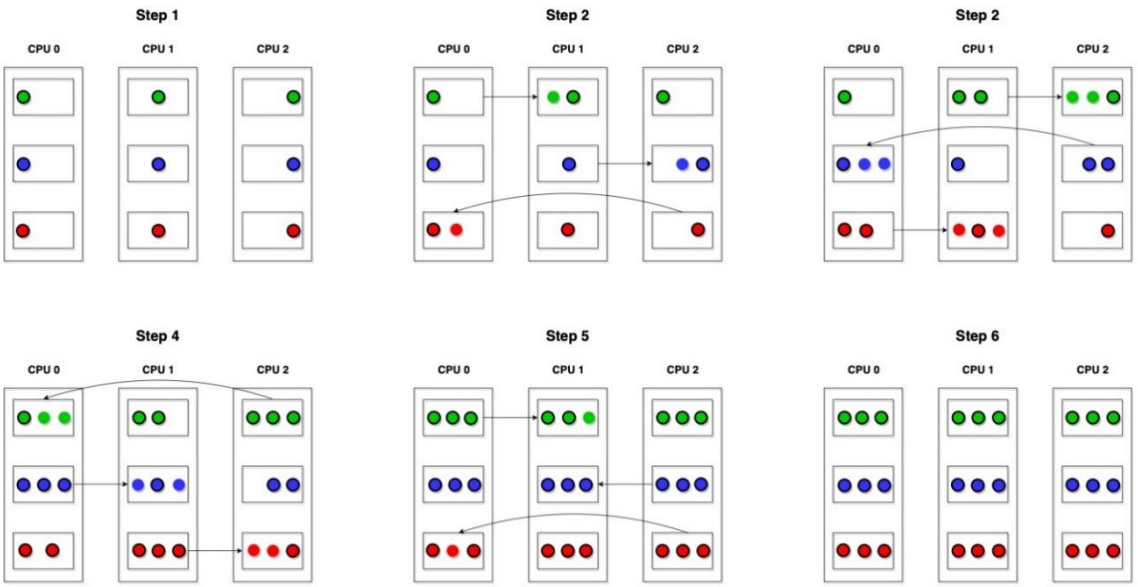

Fig. 2. All Reduce Algorithm demonstration

E. Synchronous Training:

Fig 3 shows the devices computing gradients individually. A single step of synchronous training can be illustrated with the following example, assuming, a model with two layers, each layer has two variables, and the variables are mirrored on each device, let us also assume there are two devices. In the forward pass, data is propagated through the layers. In the backward pass, the gradients for the variables are computed. At this point, the updates of the two variables on the two devices would be different as both devices would be holding different pieces of data.

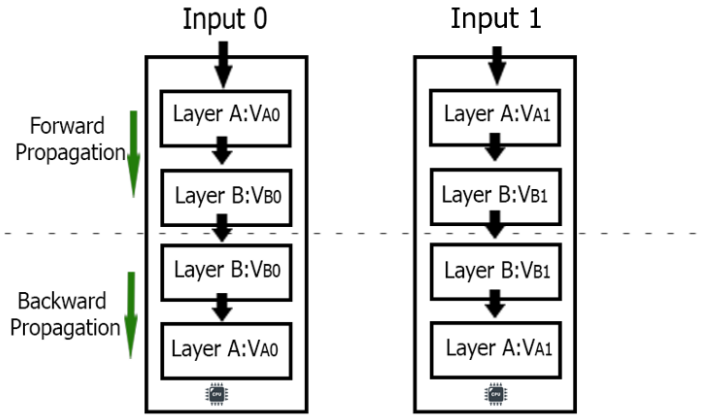

Fig. 3. Devices computing gradients individually

At that time, TensorFlow utilizes the "All Reduce" algorithm to share the updates on each device with each other, achieving a global state across two devices, as shown in Fig. 4.

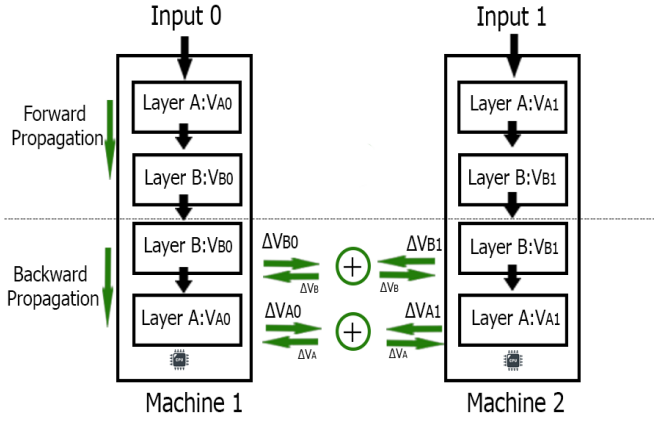

Fig. 4. Gradients aggregate between devices in lockstep

F. TensorFlow Distributed Strategies:

TensorFlow provides five APIs for the model and data parallel distributed training environment. These APIs can support several use cases for distributed training, the APIs are explained below:

1) Mirrored Strategy:

This strategy is useful when data needs to be synchronously trained across multiple GPUs in one machine. Each variable is mirrored on each GPU. The variables get updates from each other at each step and are maintained at a global state.

2) Multi Worker Mirrored Strategy:

Distributing Computation across multiple GPUs can speed up the training time by up-to N times (*Inside TensorFlow: tf.data + tf.distribute*, 2020). Where N is the number of accelerators. But there is a physical limit on the number of accelerators (Keuper and Pfreundt, 2016). To go beyond that limit, multiple machines can be utilized with each machine having one or multiple accelerators. This is possible through TensorFlow's Multi Worker Mirrored strategy.

3) TPU Strategy:

This is similar to the mirrored strategy; the main difference is that it allows All-Reduce synchronous training in TPUs (Google, 2020). Unlike the mirrored strategy, it utilizes the cross replica sum (TensorFlow, 2020) to perform the All-Reduce on TPUs. This strategy can be useful on either a single TPU or an entire pod.

4) Parameter Server Strategy:

Machines have one of two roles, parameter server task or worker task. In parameter server tasks, the global variable state is stored, they are either updated or fetched by the individual workers. Workers perform computation tasks one step at a time but not necessarily at the same rate.

5) Central Storage Strategy:

This is a special case of parameter server strategy, there is a single parameter server and its role is being fulfilled by the CPU of a machine. The CPU is connected with multiple GPUs. Each variable's copy is available on the CPU and the model is replicated across all the GPUs.

G. Federated Learning:

Federated learning where training is done on inference devices is an area of considerable interesting the research community. In Federated learning, an inference device downloads the current model and makes small changes to this model based on available data. This update is then transferred back to central service where it is averaged to improve the shared model.(Bonawitz et al.,2019) propose a scalable production system for mobile devices based upon TensorFlow (Bonawitz et al. 2019). Mohri *et al.* propose an agnostic federating learning model where the centralized model is optimized for any target distribution formed by a mixture of the client distributions (Mohri, Sivek and Suresh, 2019).

H. Decentralized Training:

The training of machine learning algorithms on distributed machines is an area of considerable interest in the research community. Lian *et al.* have shown that a distributed version of the stochastic gradient descent algorithm can achieve performance improvements of up to one order of magnitude over a centralized version of the algorithm (Lian et al., 2017). In addition, Koloskova *et al.* have shown that the performance of a distributed stotchastic gradient can be improved by utilizing a gossip based approach for communication (Koloskova, Stich and Jaggi, 2019).

I. Ensemble Models:

Ensemble models which use multiple learning algorithms to obtain better predictive performance could be used to augment this work. Zhang *et al.* proposed using ensemble models to improve optical disc segmentation (Zhang and Lim, 2020). Similarly, authors in (Heinermann and Kramer, 2016) present an ensemble model for forecasting wind-power.

III. RELATED WORK

To our knowledge, there is no directly related work that addresses cluster sizing issues in distributed training with latency constraints. However, (Keuper and Pfreundt, 2016) have provided in-depth analysis on distributed training of deep neural networks and its limitations are "communication overhead, matrix parallelization in distributed setting and distributing training data in a proper distributed fashion". Authors in (Jin et al., 2016) have discussed the convergence behaviour of multiple synchronous and asynchronous stochastic gradient descent concluding synchronous SGD provides more scaling capabilities in terms of accuracy when number of computing nodes are larger than 32. Similarly, (Geng et al., 2020) have discussed two workload parallelization strategies at clister and node-levels for reducing job completion time for deep learning workloads.

Recent work, (Natu and Ghosh, 2019) introduced the EasyDist framework. That is able, to take advantage of data parallel training, they demonstrated it using virtual machines each having a K80 GPU and 61 GB of RAM. They reduced training time by 6 times after the scaling number of Virtual Machines from 1 to 8.

In the distributed machine learning architecture context, Gaia (Hsieh et al., 2017), proposed a model that enables faster communication in Wide Area Networks (machines deployed in different datacenters) using "Approximate Synchronous Parallel" communication. This filters out information transfer between datacentres based on their significance. It only allows fewer gradient updates on the global model copy, if a worker updates gradient on a local parameter server which is above a certain significance threshold, only that update will be transferred to other data centers and will change the global gradients copy across all parameter servers across all locations. For LAN communication (machines deployed on the same datacenter), machines communicate using BSP (Bulk Synchronous Parallel) enabling fast communication between devices and taking advantage of the high bandwidth of LAN. This is illustrated in Figure 5.

In another recent work, DLion (Hong and Chandra, 2019) utilizes a micro cloud, allowing de-centralized computations and eliminating the need for parameter servers. Fig 6 shows Decentralized DLion system architecture. DLion promises a decrease in training time, increased model accuracy, and improved system scalability. DLion assigns different batch sizes to each worker according to their compute capacities. Unlike Gaia, DLion synchronizes model across devices using both Weight and Gradient exchange, which avoids accuracy reduction with the increase in cluster size. Also, DLion uses a system parameter, that decides whether to send complete or half gradients and adjusts the data size for the receiving micro cloud over the WAN depending on the available bandwidth (Network Aware Data Exchange). DLion discourages bi-directional communication between micro-clouds at the same time. However, it selects senders and receivers at a given time using probabilistic methods (based on compute capacities) over WAN at each iteration (Hong and Chandra, 2019).

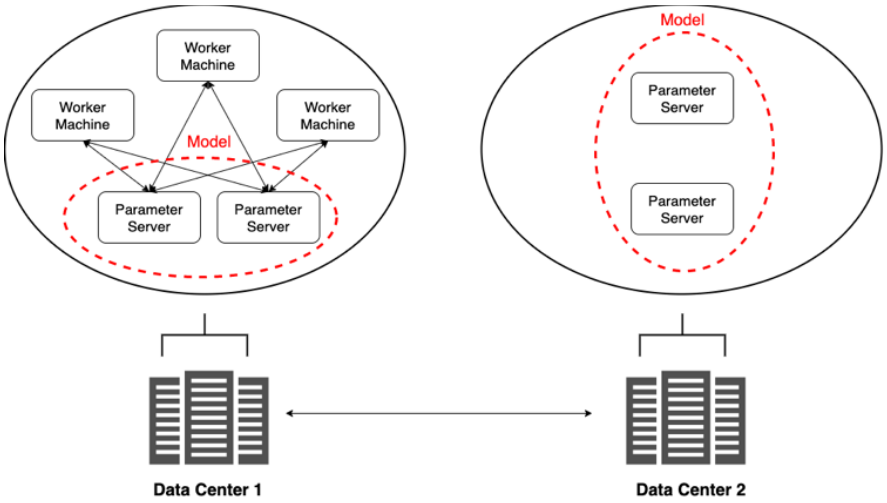

Fig. 5. Gaia system overview

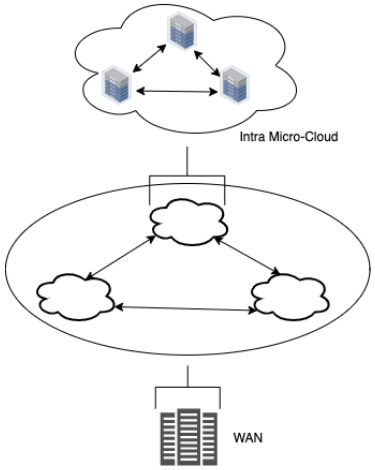

Fig. 6. Decentralized DLion system architecture.

IV. PROBLEM FORMULATION

TABLE I. Symbol Table

Symbol	Meaning
t	**Time required to execute a machine learning workload**
c	**Cost of executing a machine learning workloads**
n	**Number of machines executing a workload**
o	**Number of operations required by the machine learning workload**
x_n	**Total processing power available with n machines**
i	**Index for a particular machine in the cluster**
P_i	**Price of using the machine to execute a workload**
m	**Index for machine learning parameters choices**
s	**Index for machine learning algorithm choices**
r_m	**Number of messages required for a machine learning workload with given parameter choice**
α	**Relative price function of time**
β	**Relative price function of cost**

Let *t* be the time required to execute a machine learning workload and let *c* be the cost of executing that workload on a cloud platform. Both the time and the cost are function of the number of machines executing the workloads *n*. Time can then be represented as *t(n)*. It can be defined as follows:

$$t(n) = \frac{o}{x_n} \quad (2)$$

Where *o* is the number of operations required by the algorithm and x_n is the total processing power available with *n* machines. The cost can be represented as *c(n)*. It can be defined as follows:

$$c(n) = \sum_{i=1}^{n} p_i \quad (3)$$

Where is *i* is an index representing a machine in the cluster and p_i is the price of this machine.
The time is also a function of the networking delay between the machines in the cluster *l* and the machine learning parameters selected *m*. Thus, time can be represented as *t(n,l,m)*. This can be defined as follows:

$$t(n) = \frac{o_m}{x_n} + lr_m \quad (4)$$

Where r_m is the number of messages required by the algorithm. The time function will also change depending on the machine learning algorithm utilized *s.* Thus, time can be represented as $t_s(n,l,m)$. It can be defined as follows:

$$t_s(n) = \frac{o_{(s,m)}}{x_n} + lr_{(s,m)} \quad (5)$$

The overall goal of the work can be formulated as an optimization problem:

$$minimize \quad \alpha t_s(n,l,m) + \beta c(n) \quad \forall n,l,m \quad (6)$$
$$subject\ to \quad c(n) \leq C \quad \forall n \quad (7)$$

Where α and β are relative price weights used to indicate the importance of the cost and time goals and *C* is the maximum budget for the project which limits solutions space to computable levels. Relative price weights are a well-established methodology for optimizing a trade-off in cloud computing problems (Doyle et al.. 2011).
Finally, the overall goal can be reformulated to include all the parameters as follows:

$$minimize \quad \alpha(\frac{o_m}{x_n} + lr_m) + \beta(\sum_{i=1}^{n} p_i) \quad \forall n,l,m \quad (6)$$
$$subject\ to \quad c(n) \leq C \quad \forall n \quad (7)$$

Solving this problem is challenging due to the non-linearity of the time function and the number of variables. Thus, our solution relies on experimentation to build up a knowledge base which the system can utilize to recommend the optimal cluster size. It should be noted that our experimentation focuses on minimizing time only. This is achieved by setting $\alpha=1$ and $\beta=0$. However, any variation of the relative weight functions could be utilized in the recommendation system. The notation used in the mathematical modelling is summarized in Table I.

V. METHODOLOGY

In this section, we describe our approach for the analysis of distributed deep learning.

A. Setting up compute clusters:

1) Microsoft Azure:

A cluster is a set of at least 2 computing nodes that can send and receive data packets from each other. To set a cluster in Microsoft Azure, each Azure Virtual Machine (Microsoft, 2020) should be over the same Virtual Network and subnetwork (anavinahar, 2020). To enable distributed TensorFlow, each machine should be able to SSH to every other machine within the cluster without the need of providing keys or passwords.

a) Configuration of VM: Table 2 shows the Azure VM configuration.

TABLE II. AZURE VM CONFIGURATION

vCPUs	**4**
RAM	**8 GB**
Memory	**50 GB**
Model Number	**Standard_D2S_V3**

2) Amazon AWS:

To create a cluster in AWS, we have used EC2 instances (AWS, 2020a), that are set up over the same Virtual Private Cloud's (AWS, 2020b) private subnet. To enable distributed TensorFlow, each machine should be able to SSH to every other machine within the cluster without a key.

a) Configuration of VM : Table 3 shows the AWS EC2 configuration.

TABLE III. AWS EC2 CONFIGURATION

vCPUs	**4**
RAM	**8 GB**
Memory	**50 GB**
Model Number	**T2.Large**

B. Setting up TensorFlow:

TensorFlow was released by Google in November 2015 as a platform for Deep Learning implementations. TensorFlow can perform large scale numerical computation. It provides support for GPUs using NVIDIA cuDNN, making scaling of workloads easy on and across multiple CPUs, GPUs, and TPUs (Tensor Processing Units).

1) Keras API:

We have used Multi Worker Mirrored Strategy with Keras (Keras: the Python deep learning API, 2020), a high-level TensorFlow API that simplifies running deep learning experiments.

2) Distributed TensorFlow training strategies:

TensorFlow supports six use cases based on synchronous, asynchronous, and hardware based distributed training(TensorFlow Org., 2020). In this paper, we will be using the Multi Worker Mirrored Strategy only.

a) Multi Worker Mirrored Strategy:

TensorFlow Multi Worker Mirrored Strategy supports synchronous training on multiple GPUs across multiple machines. It creates copies of all variables in the model on each device across all workers.
Workers are run in lock-step (Poledna, 2007). Distributing computation on GPUs and CPUs across all machines. It can perform all-reduce computation and keep variables in sync on GPUs across all machines. It does so by using TensorFlow CollectiveOps (tensorflow/tensorflow, 2020) that allows data to be sent between TensorFlow workers in a broadcast fashion.

3) Network Emulation:

Creating latency on actual hardware could be costly, therefore to simulate high latency between machines and to generate packet loss and network delays in our clusters we utilized NetEm (Hemminger, 2005). We emulated latency of 20ms,40ms,60ms,80ms, and 100ms in our clusters.

VI. PERFORMANCE EVALUATION

This section covers the experimentation of our research work.

A. Cluster Size Comparison:

In table 1, we have shown Azure hardware configuration for each machine.

1) One Azure virtual machine:

We used the Sequential model on a single VM. It possesses 56,320 total parameters with a batch size of 64 and 1563 steps per epoch. The running time for each epoch was 43 seconds and each step compiled in 27ms.

2) A cluster of two Azure machines:

Now, in a distributed setting with 2 VM of the same D2S_V3 configuration, batch size set to 64 and 1563 steps per epoch. The running time for each epoch was 37s and each step compiled in 24ms. In Figure 7, the orange bar represents time per epoch on a single machine, and the blue bar represents time per epoch on the cluster (of two machines). Interestingly, under smaller batch sizes, a single machine will provide better results than two machines as the splitting the dataset and transferring gradients and biases over 2 machines will have a high amount of communication and synchronization.

3) A cluster of three AWS machines:

To scale our experiment, we opted for AWS EC2 instances. We selected three T2.Large instances having a similar configuration with Azure D2S_V3. Our goal was to compare the performance of a cluster comprised of three instances (machines) with a cluster comprised of 2 instances (machines).

This time we used the MNIST (Keras Team, 2020) dataset to evaluate the performance of our clusters with multiple batch sizes, i.e. 192, 1280, 2560, 6400, 32,000. MNIST is supervised learning algorithm but the system could be extended to unsupervised and reinforcement learning algorithms. The recommendation system is designed to provide a recommended cluster size based upon previous experimentation with the same algorithm. Augmenting the proposed system to include these

algorithms would be a matter of running on the same experimental setup with different algorithms and recording the performance so that this could be integrated into the recommendation system. This is also true for sequence models like Long short-term memory (LSTM) A supervised learning algorithm was chosen as the network overhead for these algorithms is considerable. Unsupervised algorithms like k-means are more parallelizable and thus recommendations for these algorithms are less challenging. In order to reduce cost, we chose the MNIST algorithms as it is the most challenging for recommendation systems. The results are shown in the Fig 8.

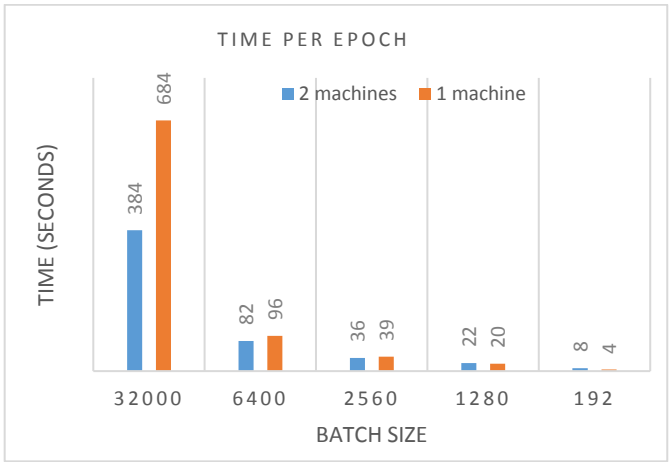

Fig. 7 Training time per epoch on 2 machines (blue) vs 1 machine (orange)

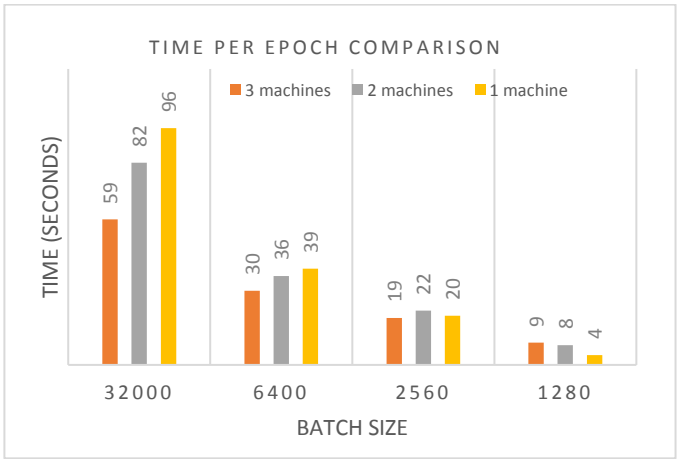

Fig. 8. Training time per epoch on 3 machines (Orange) vs 2 machines (Grey) vs 1 machine (Yellow)

The best performance is achieved with a cluster of 3 machines when the batch size is 32,000. However, in smaller batch sizes (such as 1280) we can see the single virtual machine performing twice as fast as a cluster of 2 and 3 machines. This indicates that for smaller batch sizes a single virtual machine will provide faster training speed. But as the batch size increases, clusters with a greater number of machines tend to perform better.

4) A cluster of four AWS machines:

Fig 9 shows the performance comparison of 4 cluster configurations of 1,2,3,4 machines respectively under 0 ms latency and batch sizes of 64,000, 32,000, 6400, 2560, 1280 and 192. In a cluster of 4 virtual machines at near 0 latency, a cluster of 4 virtual machines outperforms clusters of 3 virtual machines and 2 virtual machines under batch sizes of 64,000, 32,000, and 6400 by a significant margin. In batch sizes of 2560 and 1280 samples, a cluster of 3 virtual machines outperformed a cluster of 4 virtual machines and 2 virtual machines. While in the smallest batch size of 192 training samples, the single virtual machine outperformed all other cluster configurations. However, the single virtual machine failed to process a batch size of 64,000, therefore the yellow bar is missing in one of the sub-sections in the figure below. The result clearly indicates that by increasing the number of machines, the distributed training time decreases significantly given the batch size is large enough to take advantage of distributed training. In small batch sizes, it is recommended to use a single virtual machine (or a cluster with very few machines), as the benefit of splitting the small workload will not compensate the cost of communication overhead between machines.

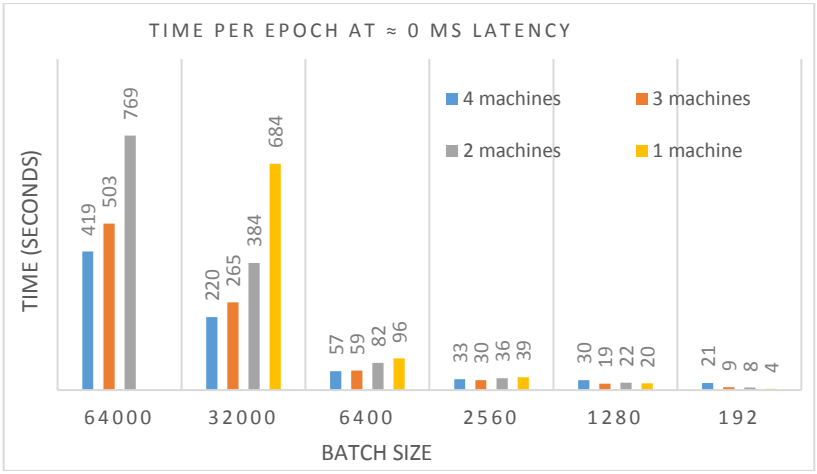

Fig. 9 Performance Comparison of 4 cluster configurations of 1,2,3,4 machines respectively under 0 ms latency and batch sizes of 64,000, 32,000, 6400, 2560, 1280 and 192

B. *Network Limitations:*

1) Effect of High Latency in Distributed Training:

One of the major challenges of distributed training is latency. Training time can drastically increase due to minor latency issues between worker nodes. To demonstrate this, we trained CIFAR10 (CIFAR-10, 2020) dataset with a batch size of 64 using the Sequential model. With a time per epoch of 37 seconds at 0 milliseconds. Then we simulated latency ranging from 20ms, 40ms, 60ms, 80ms, and 100ms. We saw a significant drop in the performance of our cluster under high latency conditions as shown in Fig. 10.

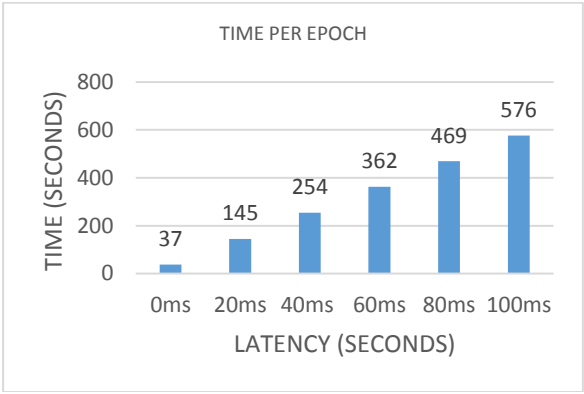

Fig. 10. Increase in training time with an increase in latency between machines. Trained on a cluster of 2 Azure Virtual Machines

2) Detailed Performance Analysis of Distributed Training Under Several Latency Constraints:

a) 20 ms Latency:

For further experiments, we used the MNIST dataset. We introduced 20ms latency between each virtual machine. The results are shown in the Figure 11. In a batch size of 64,000, the Single virtual machine configuration failed to handle the huge batch size and the program resulted in an error. While a cluster of 3 machines achieved the fastest training with 4.2% faster training than the cluster of 4 virtual machines and 31% faster than a cluster of 2 virtual machines. This pattern changes in smaller batch sizes. For example, at the 6400 batch size, a cluster of 4 virtual machines achieved 14% and 38% faster training speed than a cluster of 3 machines and 2 machines, respectively. In a batch size of 2560, again a cluster of 4 machines outperform clusters of 3 and 2 machines. However, under a very small batch size of 192 samples, we can see a single virtual machine outperforming clusters of 2,3 and 4 machines.

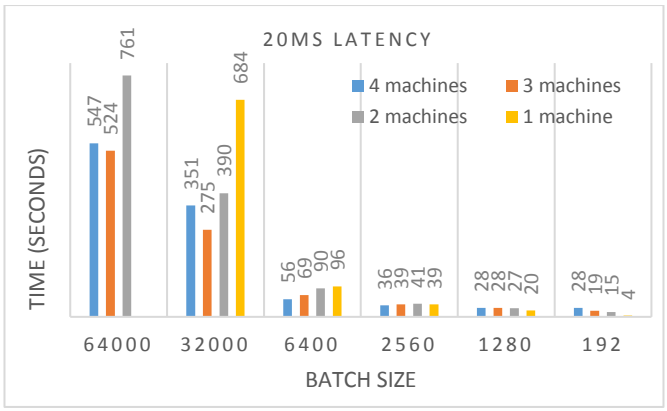

Fig. 11. Time per epoch comparison 4 cluster configurations, consisting of 1,2,3,4 machines respectively under 20 ms latency and several batch sizes.

As it is evident with the results that, with the higher batch sizes (64,000 and 32,000), a cluster of 3 machines is performing better than a cluster of 4 machines under 20ms latency. Previously, we got opposite results under 0ms latency as shown in Figure 9.

b) 40 ms Latency:

For simplicity, we are considering only three clusters in this example and onwards (we are not including the single machine, as it fails to run the batch size of 64,000 samples). Fig 12 shows the time per epoch comparison 3 cluster configurations, consisting of 2,3,4 machines respectively under 40 ms latency and several batch sizes. At 40ms latency between virtual machines, we can see a similar pattern as before. In batch sizes of 64,000 and 32,000, a cluster of 3 machines provides faster training speeds than others. In batch sizes of 6400 and 2560, a cluster of 4 machines provide faster training speed than 3 machines and 2 machines. In batch sizes of 1280 and 192, a single virtual machine performs better than clusters of 2, 3, or 4 machines.

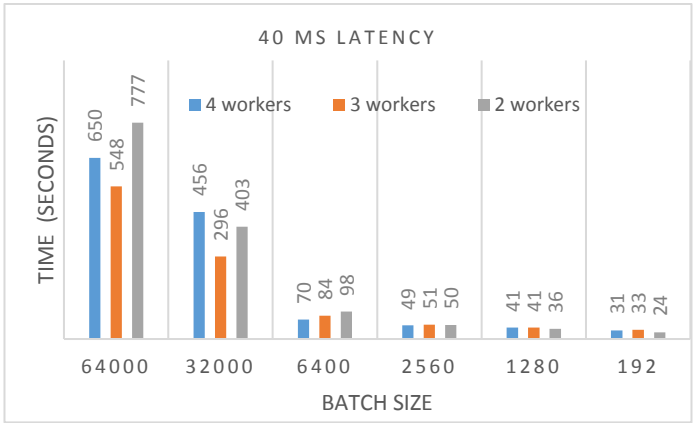

Fig. 12, Time per epoch comparison 3 cluster configurations, consisting of 2,3,4 machines respectively under 40 ms latency and several batch sizes.

c) 60 ms Latency:

Fig 13 shows the time per epoch comparison 3 cluster configurations, consisting of 2,3,4 machines respectively under 60 ms latency and several batch sizes.

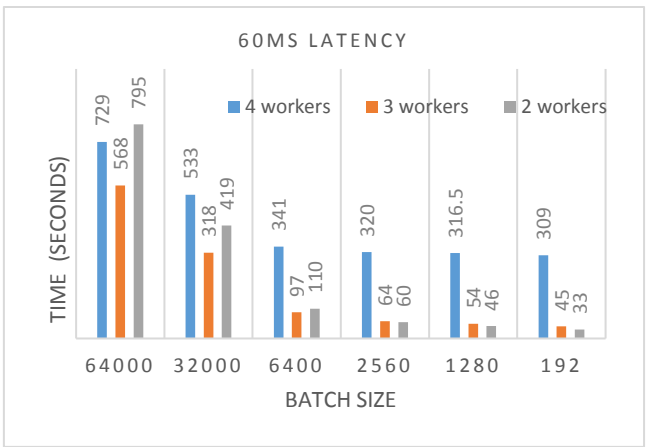

Fig. 13. Time per epoch comparison 3 cluster configurations, consisting of 2,3,4 machines respectively under 60 ms latency and several batch sizes.

At higher latency such as 60ms, a cluster of 4 machines is not able to take advantage of distributed training. It does distribute workload, but due to communication delays between machines, the benefit of distributing workload is not advantaged by the cost of slower gradient aggregation between machines due to slower network traffic and high latency. Furthermore, a cluster of 3 virtual machines outperforms a cluster of 2 machines under batch sizes of 64,000, 32,000, and 6400. However, under smaller batch sizes, a cluster of 2 machines achieves faster training speed than a cluster of 3 machines.

d) 80 ms Latency:

Fig 14 shows the time per epoch comparison 3 cluster configurations, consisting of 2,3,4 machines respectively under 80 ms latency and several batch sizes. At 80 ms latency, the patterns are quite similar to the previous experiment. A cluster of 3 machines provides faster computation speed in batch sizes of 64,000, 32,000, and 6400. While a cluster of 2 machines provide faster computation time in batch sizes of 2560, 1280, and 192. A cluster of 4 machines performs poorly in all batch sizes.

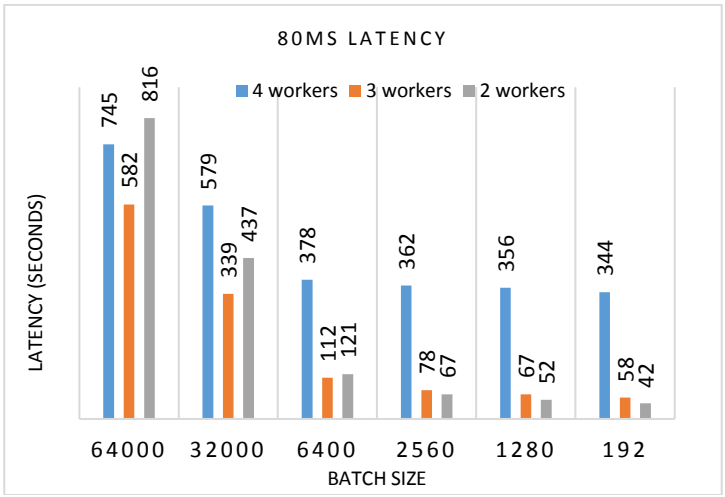

Fig. 14. Time per epoch comparison 3 cluster configurations, consisting of 2,3,4 machines respectively under 80 ms latency and several batch sizes.

e) 100 ms Latency:

Fig 15 shows the time per epoch comparison 3 cluster configurations, consisting of 2,3,4 machines respectively under 100 ms latency and several batch sizes. At 100ms latency, the pattern is quite similar to the previous experiment. A cluster of 3 machines provides faster computation speed in batch sizes of 64,000, 32,000, and 6400. While a cluster of 2 machines provide faster computation time in batch sizes of 2560, 1280, and 192. However, taking a closer look at the batch size of 6400, there is a minor difference between the performance of a cluster of 2 machines and a cluster of 3 machines. This indicates that increasing latency further would show a performance drop for a cluster of 3 machines, while cluster of 2 machines would not suffer from the communication delays as much as the alternatives.

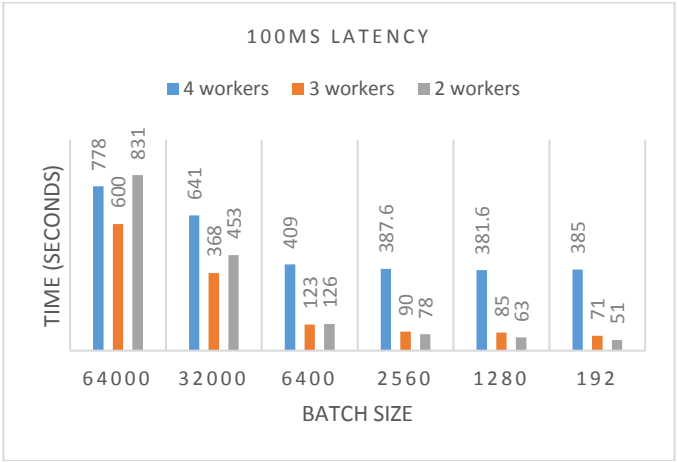

Fig. 15. Time per epoch comparison 3 cluster configurations, consisting of 2,3,4 machines respectively under 100 ms latency and several batch sizes.

VII. Experimental Results

We have evaluated the performance of clusters of 3 different sizes along with a single virtual machine running a deep learning model. We have observed that under ideal latency conditions distributing training over multiple machines massively improves the training time of deep learning models given batch sizes are significantly large enough to take complete advantage of distributed architecture. Under smaller batch sizes, clusters with fewer machines outperform clusters with more

machines. In high latency conditions, we have observed small clusters outperforming larger clusters, as distributed training speedup is not compensated by the communication overhead.

A. Our Model:

With the help of our rigorous experimentation, we collected 114 data points with batch sizes 64,000,32,000,6400,1280 and 192. Clusters ranging from 1,2,3 to 4 machines. Latency ranging from 0,20,40,60,80 milli-seconds. We recorded Time Per Epoch in each scenario. Table 4 shows the database.

TABLE IV. DATABASE (FEW ROWS INCLUDED FOR ILLUSTRATION)

Workers	Batch Size	Latency	Time Per Epoch
4	64,000	0	419
4	32,000	0	220
4	6400	0	57
3	64,000	0	503
3	32,000	0	265
2	6400	20	90
2	2560	20	41
1	6400	0	96
..	..	..	..
1	2560	0	39

Utilizing our data, we have developed a recommendation system that is able to recommend the optimum cluster size for distributed deep learning environments. Fig 16 shows the landing page of the API

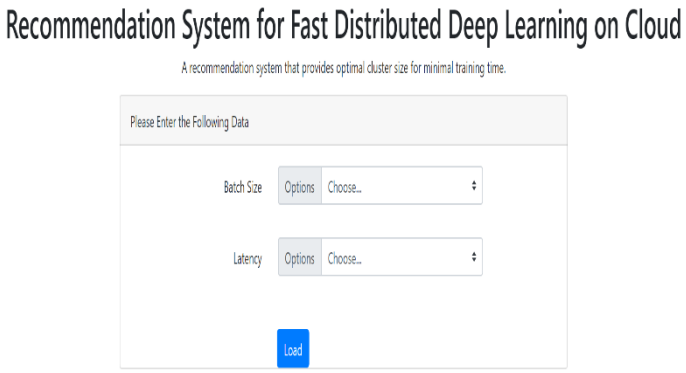

Fig. 16. Landing page of the API

When a user enters latency and Batch Size, our model simply looks for all the rows in the database having that latency and batch size, then it filters it by minimum Time per epoch and returns the respective number of workers (virtual machines). We have made our model publicly available using a REST API, Users can enter Batch Size and network latency of their environment. Our model will use simple look-up techniques and will return the optimum number of virtual machines (worker nodes) that are able to train the deep learning model in minimal time. Fig 17 shows the API response on an input by a user.

As an example, given a batch size of 64,000 and 0 latency, our model recommends a cluster size of 4 virtual machines. Similarly, when given a batch size of 64,000 samples and 20 ms latency, it recommends a cluster size of 3 virtual machines. Fig 18 shows an API response on another input by a user.

VIII. CONCLUSIONS AND FUTURE WORK

There is an increasing demand for high performing distributed systems for modern machine learning applications. Existing research does not address cluster sizing techniques for faster training of deep learning models in environments having network limitations (high latency between machines) in cases when machines are geographically distributed from each other. In this paper, we have discussed the impact of cluster size (number of machines connected to each other) in training deep learning models for multiple batch sizes under multiple latencies. We concluded our research by introducing a recommendation system that can predict the best cluster size for the fastest training time, for a given latency and batch size.

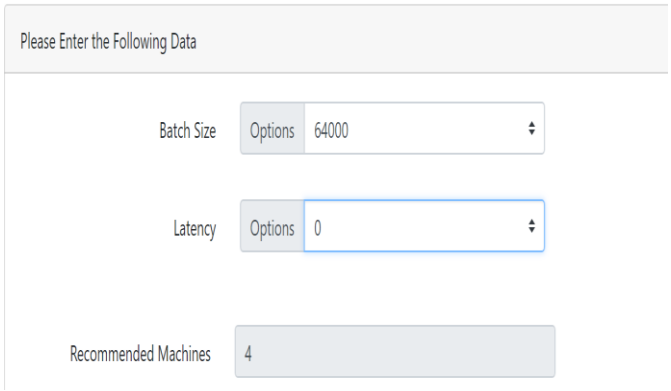

Fig. 17. API response on an input by a user.

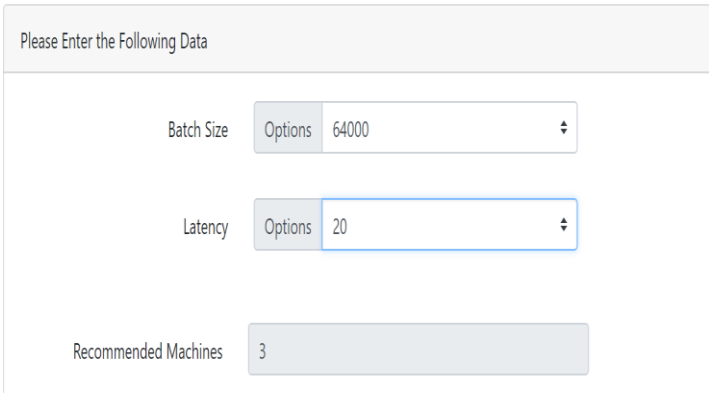

Fig. 18. API response on another input by a user.

As part of future works, this research can be expanded in the below directions.

- Explore how to integrate our system with frameworks like iThermoFog (Tuli et al. 2020) to dynamically adapt in volatile workload based environments
- Our proposed model of fast distributed deep learning is based on a limited number of distributed environments. We have simulated latency of 20ms,40ms,60ms,80ms, and 100ms. The data is recorded utilizing two versions of virtual machines, one from AWS and another from Azure. In practice, there are hundreds of virtual machines available from commercial cloud providers. Making a robust cluster size recommendation system would require hours of training on different types of virtual machines under hundreds of environments.
- We have recorded data using the Sequential model trained on the Mnist dataset. In practice, the data needs to be collected by training several models. The training time and behaviour of each model under multiple environments could then be recorded.
- We have utilized a simple look up technique for a cluster size recommendation system as we have a limited number of data points. In future research, the model could be trained using machine learning algorithms such as Random Forest or Reinforcement Learning.
- We have not considered accuracy as a factor while considering the optimal number of recommended worker nodes. In future research work, we aim to incorporate accuracy as a factor, so that the model should recommend cluster size with minimal training time and highest accuracy. In the experiments, no accuracy decrease has been observed after scaling the size of clusters. However, significant testing is required to justify this claim.
- We have not considered cost as a factor for cluster sizing. For instance, the model might be choosing a cluster configuration with a higher number of machines, without considering the cost associated with it. In future work, we will consider the cost of each VM to make cost aware decisions.
- The core idea of this project is parallelizing SGD and using all-reduce algorithm to achieve global states. A downside comes with distributed SGD, where workers during the communication part stay idle. In the future, we aim to adopt Decentralized processing as shown in (Hong and Chandra, 2019).
- The recommender system could also be augmented to incorporate semantic split neural network models. These models reduce network overhead by splitting weights into a set or hierarchy. Thus, the network overhead would be reduced and a larger cluster may offer better performance.

Declarations

a. Funding

Funding information is not applicable

b. Conflicts of interest

On behalf of all authors, the corresponding author states that there is no conflict of interest.

c. Availability of data and material

Data and material is available here

https://github.com/iamhassaan/Fast_DL

https://github.com/iamhassaan/Fast_Distributed_DL

d. Code availability

Software application code is available here

https://github.com/iamhassaan/Thesis_API

e. Authors' contributions

Muhammad Hassaan Anwar (Conceptualization: Lead; Data curation: Lead; Formal analysis: Lead; Funding acquisition: Lead; Investigation: Lead; Methodology: Lead; Software: Lead; Validation: Lead; Writing – original draft: Lead)

Saeid Ghafouri (Conceptualization: Supporting; Data curation: Supporting; Writing – original draft: Supporting)

Sukhpal Singh Gill (Supervision: Supporting; Writing – review & editing: Supporting)

Joseph Doyle (Conceptualization: Supporting; Methodology: Supporting; Supervision: Lead; Validation: Supporting; Writing – review & editing: Lead)

REFERENCES

anavinahar. (2020). Azure Virtual Network. Available from https://docs.microsoft.com/en-us/azure/virtual-network/virtual-networks-overview [Accessed 25 August 2020].

Annamalai, Suresh, R. Udendhran, and S. Vimal. "An intelligent grid network based on cloud computing infrastructures." In Novel Practices and Trends in Grid and Cloud Computing, pp. 59-73. IGI Global, 2019.

Annamalai, Suresh, R. Udendhran, and S. Vimal. "Cloud-based predictive maintenance and machine monitoring for intelligent manufacturing for automobile industry." In Novel Practices and Trends in Grid and Cloud Computing, pp. 74-89. IGI Global, 2019.

AWS. (2020a). Amazon EC2. Available from https://aws.amazon.com/ec2/ [Accessed 25 August 2020].

AWS. (2020b). Amazon Virtual Private Cloud (VPC). Available from https://aws.amazon.com/vpc/ [Accessed 25 August 2020].

Bonawitz, K. et al. (no date). Towards Federated Learning at Scale: System Design. *System Design*, 15.

Bottou, L. (2010). Large-Scale Machine Learning with Stochastic Gradient Descent. 10.

Chen, T. et al. (2016). MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. 6.

CIFAR-10. (2020). Available from https://www.cs.toronto.edu/~kriz/cifar.html [Accessed 31 August 2020].

Doyle, J., O'Mahony, D. and Shorten, R., 2011, August. Server selection for carbon emission control. In *Proceedings of the 2nd ACM SIGCOMM workshop on Green networking* (pp. 1-6).

Feng, M., Xiang, B. and Zhou, B. (2016). Distributed Deep Learning for Question Answering. *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 24 October 2016. Indianapolis Indiana USA: ACM, 2413–2416. Available from https://doi.org/10.1145/2983323.2983377 [Accessed 24 August 2020].

Geng, X. et al. (2020). Interference-aware parallelization for deep learning workload in GPU cluster. *Cluster Computing*, 23 (4), 2689–2702. Available from https://doi.org/10.1007/s10586-019-03037-6.

Gill, S. S., Tuli, S., Xu, M., et al. (2019). Transformative effects of IoT, Blockchain and Artificial Intelligence on cloud computing: Evolution, vision, trends and open challenges. Internet of Things, 8, 100118.

Gill, S. S., Tuli, S., Toosi, A. N., Cuadrado, F., Garraghan, P., Bahsoon, R., & Buyya, R. (2020). ThermoSim: Deep learning based framework for modeling and simulation of thermal-aware resource management for cloud computing environments. Journal of Systems and Software, 166, 110596.

Goyal, P. et al. (2018). Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv:1706.02677 [cs]*. Available from http://arxiv.org/abs/1706.02677 [Accessed 23 August 2020].

Heinermann, J. and Kramer, O. (2016). Machine learning ensembles for wind power prediction. *Renewable Energy*, 89, 671–679. Available from https://doi.org/10.1016/j.renene.2015.11.073.

Hemminger, S. (2005). Network Emulation with NetEm. 9.

Hong, R. and Chandra, A. (2019). DLion: Decentralized Distributed Deep Learning in Micro-Clouds. 9.

Hsieh, K. et al. (2017). Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds. 21.

Inside TensorFlow: tf.data + tf.distribute. (2020). Available from https://www.youtube.com/watch?v=ZnukSLKEw34 [Accessed 24 August 2020].

Jin, P.H. et al. (2016). How to scale distributed deep learning? *arXiv:1611.04581 [cs]*. Available from http://arxiv.org/abs/1611.04581 [Accessed 10 April 2021].

Keras Team, K. (2020). Keras documentation: MNIST digits classification dataset. Available from https://keras.io/api/datasets/mnist/ [Accessed 24 August 2020].

Keras: the Python deep learning API. (2020). Available from https://keras.io/ [Accessed 24 August 2020].

Keuper, J. and Pfreundt, F.-J. (2016). Distributed Training of Deep Neural Networks: Theoretical and Practical Limits of Parallel Scalability. *arXiv:1609.06870 [cs]*. Available from http://arxiv.org/abs/1609.06870 [Accessed 24 August 2020].

Koloskova, A., Stich, S.U. and Jaggi, M. (no date). Decentralized Stochastic Optimization and Gossip Algorithms with Compressed Communication. 10.

Lian, X. et al. (no date). Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent. 11.

Mahajan, K., Balasubramanian, A., Singhvi, A., Venkataraman, S., Akella, A., Phanishayee, A., & Chawla, S. (2020). Themis: Fair and Efficient {GPU} Cluster Scheduling. In 17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20) (pp. 289-304

Microsoft. (2020). Virtual Machines (VMs) for Linux and Windows | Microsoft Azure. Available from https://azure.microsoft.com/en-us/services/virtual-machines/ [Accessed 25 August 2020].

Mohri, M., Sivek, G. and Suresh, A.T. (no date). Agnostic Federated Learning. 30.

Moritz, P. et al. (2016). Ray: A Distributed Framework for Emerging AI Applications. 18.

Natu, V. and Ghosh, R. (2019). EasyDist: An End-to-End Distributed Deep Learning Tool for Cloud. *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data - CoDS-COMAD '19*. 2019. Kolkata, India: ACM Press, 265–268. Available from https://doi.org/10.1145/3297001.3297037 [Accessed 23 August 2020].

Poledna, S. (2007). *Fault-Tolerant Real-Time Systems*. Springer Science & Business Media.

Recht, B. et al. (2011). Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. 9.

Simonyan, K. and Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv:1409.1556 [cs]*. Available from http://arxiv.org/abs/1409.1556 [Accessed 24 August 2020].

TensorFlow. (2020). tf.distribute.StrategyExtended | TensorFlow Core v2.3.0. *TensorFlow*. Available from https://www.tensorflow.org/api_docs/python/tf/distribute/StrategyExtended [Accessed 26 August 2020].

TensorFlow Org. (2020). Distributed training with TensorFlow | TensorFlow Core. *TensorFlow*. Available from https://www.tensorflow.org/guide/distributed_training [Accessed 24 August 2020].

tensorflow/tensorflow. (2020). *GitHub*. Available from https://github.com/tensorflow/tensorflow [Accessed 24 August 2020].

Tuli, Shreshth, Sukhpal S. Gill, Giuliano Casale, and Nicholas R. Jennings. "iThermoFog: IoT - Fog based automatic thermal profile creation for cloud data centers using artificial intelligence techniques." Internet Technology Letters 3, no. 5 (2020): e198

Tuli, Shreshth, Shivananda Poojara, Satish N. Srirama, Giuliano Casale, and Nicholas R. Jennings. "COSCO: Container Orchestration using Co-Simulation and Gradient Based Optimization for Fog Computing Environments." arXiv preprint arXiv:2104.14392 (2021).

Veeramanikandan et al. (2020). Data Flow and Distributed Deep Neural Network based low latency IoT-Edge computation model for big data environment. *Engineering Applications of Artificial Intelligence*, 94, 103785. Available from https://doi.org/10.1016/j.engappai.2020.103785.

Xu, L., Xu, M., Semmes, R., Li, H., Mu, H., Gui, S., ... & Buyya, R. (2020). A Reinforcement Learning Based Approach to Identify Resource Bottlenecks for Multiple Services Interactions in Cloud Computing Environments. In International Conference on Collaborative Computing: Networking, Applications and Worksharing (pp. 58-74). Springer, Cham.

Zhang, L. and Lim, C.P. (2020). Intelligent optic disc segmentation using improved particle swarm optimization and evolving ensemble models. *Applied Soft Computing*, 92, 106328. Available from https://doi.org/10.1016/j.asoc.2020.106328.

Zhang, Z. et al. (2018). A Quick Survey on Large Scale Distributed Deep Learning Systems. *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. December 2018. Singapore, Singapore: IEEE, 1052–1056. Available from https://doi.org/10.1109/PADSW.2018.8644613 [Accessed 24 August 2020].

Authors Biography

Muhammad Hassaan Anwar is an MSc student at Queen Mary University of London. Currently, he is completing his industrial placement at an energy-consulting startup, working as a commercial Data Engineer. His research interests include IoT, Machine Learning, Distributed Systems and Cloud Computing.

Saeid Ghafouri received his M.Sc. in the field of Artificial Intelligence from the Computer Engineering Department, K. N. Toosi, University of Technology, Tehran, Iran. He is currently pursuing a Ph.D. degree at the Queen Mary University of London advised by Dr. Joseph Doyle. His current research interests include cloud computing, resource allocation in cloud computing, machine learning applications, and deep reinforcement learning.

Sukhpal Singh Gill is a Lecturer (Assistant Professor) in Cloud Computing at School of Electronic Engineering and Computer Science, Queen Mary University of London, UK. Prior to this, Dr. Gill has held positions as a Research Associate at the School of Computing and Communications, Lancaster University, UK and also as a Postdoctoral Research Fellow at CLOUDS Laboratory, The University of Melbourne, Australia. Dr. Gill is serving as an Associate Editor in IET Networks Journal. His research interests include Cloud Computing, Fog Computing, Software Engineering, Internet of Things and Healthcare. For further information, please visit http://www.ssgill.me

Joseph Doyle graduated from Trinity College Dublin in 2009 with a B.A.I., B.A. degree in Computer and Electronic Engineering as a gold medalist. He was awarded a Ph.D in 2013 from Trinity College Dublin. He was a post-doctoral researcher in Trinity College Dublin and University College London from 2013 to 2014 and 2014 to 2016, respectively. He was Senior Lecturer in the University of East London, London, U.K. from 2016 to 2018. He is a co-founder of Dithen Ltd. (London, U.K.) and is also Lecturer at Queen Mary University of London, London, U.K. He is an RSE cloud fellow. His research interests include cloud computing, virtual machine classification, green computing, and network optimization.