Comparison & study of Distributed Deep Learning Training Techniques

Amey Bhilegaonkar
abhilega@asu.edu

Gaurav Hoskote
ghoskote@asu.edu

Ninad Nale
nnale@asu.edu

ABSTRACT

With the rise of large language and multimodal models, the complexity and size of deep learning models have proved traditional single-machine training increasingly expensive and time-consuming. This limitation impedes advancements in various AI applications like image recognition, natural language processing, and scientific discovery. In this context, various distributed techniques for deep learning, such as data parallelism and model parallelism, offer a promising solution. This project delves into an explorative study of different strategies in implementing data parallelism and model parallelism, the two primary distributed deep learning training techniques. Our objective is to gain a comprehensive understanding of their theoretical underpinnings, practical considerations, and best practices for implementation. The results demonstrate that using these strategies has significant advantages over the traditional training approaches.

KEYWORDS

Data Parallelism, Model Parallelism, Parameter Server, TFServe, MLops, Vertex AI, PyTorch

1. INTRODUCTION

Over recent years, machine learning models have been used on a larger scale by organizations to find business insights by leveraging their data. These organizations find training large and complex models on traditional single-machine setups slow and compute-heavy. Hence the problem under the investigation here is that training large and complex machine learning models on traditional single-machine setups is slow and computationally expensive, hindering organizations from leveraging them for large-scale data analysis and business insights.

To overcome these bottlenecks, distributed deep learning techniques like Data parallelism and Model parallelism can be leveraged. Both techniques significantly accelerate training. Data parallelism distributes the training dataset across multiple machines, allowing each to train on a portion concurrently. This effectively trains the model multiple times simultaneously, leading to faster overall training. Model parallelism, on the other hand, splits the model itself across machines, enabling parallel computation of different parts. This is particularly beneficial for very large models that wouldn't fit on a single machine's memory. This harnesses the power of multiple processing units. However, the real-world implementation of these paradigms requires a nuanced understanding and adaptation based on specific model architectures, datasets, and hardware configurations. By investigating the trade-offs between data and model parallelism, this project equips researchers and practitioners with valuable insights for selecting the optimal training strategy based on their specific needs. This report will help gain more understanding into the potential to accelerate the development and deployment of cutting-edge AI models, fostering advancements across various scientific and technological disciplines.

This report is structured as follows. Section 2 presents a comprehensive literature survey, summarizing the theoretical underpinnings and practical considerations of data and model parallelism. Section 3 details the chosen dataset for our experimentation. Section 4 delves into the methodology employed, including the experimental setup (Section 4.1). Section 5 presents the results obtained from our experiments, followed by a discussion of their implications. Finally, Section 6 concludes the report by summarizing our findings and outlining potential avenues for future work.

2. LITERATURE SURVEY

This section discusses the related works that we have studied to understand the concepts required for the implementation of this project. We will begin with fundamental information about each distributed training paradigm studied, and advance towards various strategies in tensorflow.

Model parallelism is a technique used to distribute the training of a large model across multiple machines. [2] discusses two distributed optimization algorithms that can be used with model parallelism: Downpour SGD and Sandblaster L-BFGS. Downpour SGD is an asynchronous variant of stochastic gradient descent (SGD) that allows multiple model replicas to update the parameters of a model simultaneously. Sandblaster L-BFGS is a distributed implementation of the L-BFGS algorithm, which is a batch optimization method. The Downpour SGD with the Adagrad adaptive learning rate procedure was the most effective method for training deep learning models on a limited computational budget [2].
Data parallelism is a technique to improve the execution time of data-intensive workflows by distributing data processing tasks across multiple computing resources. [3] proposes a method to automatically parallelize workflow activities based on annotations that describe how activities access and consume data. The annotations are used to identify opportunities for data parallelism, such as when activities process data objects independently or in groups. The workflow model is then redesigned to exploit these opportunities by creating replicas of activities and distributing the input data among them. The size of batches in data parallelism affects the training time and accuracy. [5] investigates the impact of data parallelism on training time. The findings demonstrate a three-stage pattern: perfect scaling where doubling batch size halves training steps, diminishing returns where the benefit lessens, and finally reaching a maximum data parallelism point where further increase offers no improvement [5]. The study also explores how factors like model architecture and optimizer selection influence this relationship, providing valuable insights for practitioners aiming to optimize neural network training through data parallelism [1].
TensorFlow Distributed Training Strategies

TensorFlow offers various distributed training strategies to facilitate efficient training on multiple machines. Here, we discuss three relevant strategies for distributed training: Mirrored Strategy, Parameter Server Strategy, and Multi-Worker Mirrored Strategy:

Mirrored strategy is well-suited for synchronous distributed training on a single machine with multiple GPUs. It creates a replica of the model and its variables on each available GPU device. These replicas, known as MirroredVariables, are kept in sync through communication and update aggregation techniques like all-reduce [6].

Parameter Server Strategy caters to both synchronous and asynchronous distributed training scenarios across multiple machines. It employs a dedicated set of machines as parameter servers, responsible for storing and managing the model's variables. Worker machines, containing replicas of the model architecture, perform computations on their local datasets and communicate updates to the parameter servers. This approach allows for asynchronous training as workers can proceed with computations without waiting for all workers to complete a step.

Multi-worker mirrored strategy extends the Mirrored Strategy concept to a multi-worker setting. It enables synchronous distributed training across multiple workers, where each worker can have multiple GPUs. Similar to the Mirrored Strategy, each worker maintains a replica of the model and its variables. However, communication and synchronization occur across all worker machines for efficient training [4].

3. DATASET

In this project we have used the MNIST dataset, a widely recognized benchmark for evaluating image classification algorithms. The dataset consists of around 70,000 images of handwritten digits (0-9) from various individuals. Each image is grayscale, centered, and normalized to a fixed size of 28x28 pixels.
The reasons why MNIST was chosen as the dataset for this project:

Simplicity and Standardization: The dataset provides a well-understood and standardized set of images for classification tasks. The relatively smaller size and low dimensionality make it computationally efficient to train and experiment with various distributed training techniques. This allows us to focus on the core concepts of data and model parallelism without introducing complexities associated with very large datasets.

Focus on Techniques: By using MNIST, we can prioritize understanding the theoretical underpinnings and practical considerations of data and model parallelism. The relatively smaller size allows for faster experimentation and clearer observation of performance differences between the two approaches. This focus on the techniques themselves is crucial for our project goals.

While the MNIST dataset might not represent the real-world complexities of large-scale image recognition tasks, it provides an ideal platform for our initial exploration of data and model parallelism. The insights gained from this project will lay the groundwork for tackling more intricate datasets and real-world applications of distributed deep learning in the future.

4. METHODOLOGY

MirroredStrategy: This strategy is suitable for data parallelism, where the model is replicated across multiple workers, and each worker processes a different subset of the data. It uses all-reduce to aggregate gradients across workers, enabling synchronous training.

ModelParallelStrategy: This strategy is used for model parallelism, where different parts of the model are placed on different devices (e.g., GPUs). It allows you to partition the model graph across multiple devices to train large models that do not fit on a single device.

ParameterServerStrategy: This strategy uses a parameter server architecture, where model parameters are stored on parameter servers, and workers fetch and update the parameters during training. This can be useful for large models that do not fit a single worker.

5. Experimental Setup

Set up the Vertex AI Notebook environment:

- Create a new Vertex AI Notebook instance with the appropriate machine type (e.g., Tesla GPU).
- Install the necessary TensorFlow and distributed training libraries in the notebook.

Implement data parallelism using MirroredStrategy:

- Define the model architecture and training pipeline.

- Wrap the model in the MirroredStrategy and configure the distributed training setup.
- Measure and compare the training speed, convergence, and other performance metrics with single-worker training.

Explore model parallelism using ModelParallelStrategy:

- Explore the partitioning of the model graph across multiple devices using the ModelParallelStrategy.
- Study how it manages the communication and synchronization between the partitioned model components.
- Evaluate the performance of model parallelism, considering factors like communication overhead and computational efficiency.

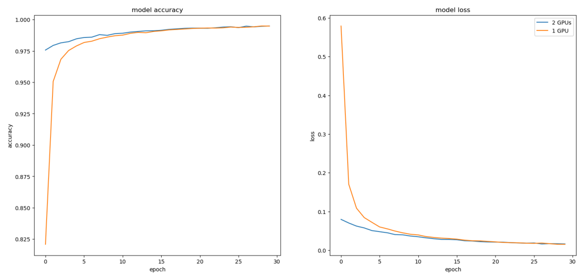
Explore the ParameterServerStrategy:

- Explore the parameter server architecture, with parameter servers and worker nodes.
- Go through the documentation for the training pipeline using the ParameterServerStrategy.
- Analyze the performance and scalability of the parameter server approach, especially for large models.

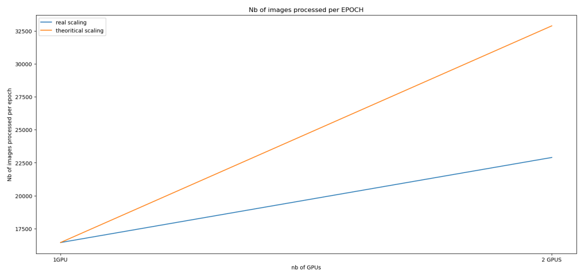
Document Best Practices:

- Summarize the key learnings and insights gained from the experiments.
- Provide practical recommendations for selecting the optimal parallelism strategy based on factors like model size, dataset characteristics, and training objectives.

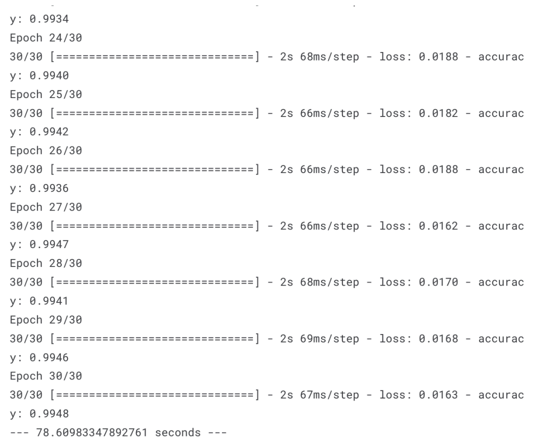
6. EVALUATION RESULTS



Model Accuracy and Model Loss for 1 GPU vs 2 GPU

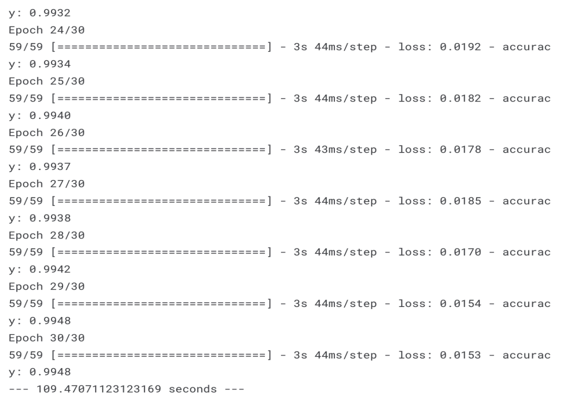


Theoretical Scaling vs Real Scaling



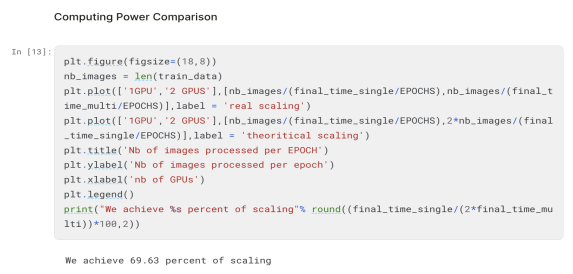
```
y: 0.9934
Epoch 24/30
30/30 [==============================] - 2s 68ms/step - loss: 0.0188 - accurac
y: 0.9940
Epoch 25/30
30/30 [==============================] - 2s 66ms/step - loss: 0.0182 - accurac
y: 0.9942
Epoch 26/30
30/30 [==============================] - 2s 66ms/step - loss: 0.0188 - accurac
y: 0.9936
Epoch 27/30
30/30 [==============================] - 2s 66ms/step - loss: 0.0162 - accurac
y: 0.9947
Epoch 28/30
30/30 [==============================] - 2s 68ms/step - loss: 0.0170 - accurac
y: 0.9941
Epoch 29/30
30/30 [==============================] - 2s 69ms/step - loss: 0.0168 - accurac
y: 0.9946
Epoch 30/30
30/30 [==============================] - 2s 67ms/step - loss: 0.0163 - accurac
y: 0.9948
--- 78.60983347892761 seconds ---
```

Training time for 2 GPUs 78.6 s



```
y: 0.9932
Epoch 24/30
59/59 [==============================] - 3s 44ms/step - loss: 0.0192 - accurac
y: 0.9934
Epoch 25/30
59/59 [==============================] - 3s 44ms/step - loss: 0.0182 - accurac
y: 0.9940
Epoch 26/30
59/59 [==============================] - 3s 43ms/step - loss: 0.0178 - accurac
y: 0.9937
Epoch 27/30
59/59 [==============================] - 3s 44ms/step - loss: 0.0185 - accurac
y: 0.9938
Epoch 28/30
59/59 [==============================] - 3s 44ms/step - loss: 0.0170 - accurac
y: 0.9942
Epoch 29/30
59/59 [==============================] - 3s 44ms/step - loss: 0.0154 - accurac
y: 0.9948
Epoch 30/30
59/59 [==============================] - 3s 44ms/step - loss: 0.0153 - accurac
y: 0.9948
--- 109.47071123123169 seconds ---
```

Training time for 1 GPU 109.5 s



Computing Power Comparison

```
In [13]:
plt.figure(figsize=(18,8))
nb_images = len(train_data)
plt.plot(['1GPU','2 GPUS'],[nb_images/(final_time_single/EPOCHS),nb_images/(final_t
ime_multi/EPOCHS)],label = 'real scaling')
plt.plot(['1GPU','2 GPUS'],[nb_images/(final_time_single/EPOCHS),2*nb_images/(final
_time_single/EPOCHS)],label = 'theoritical scaling')
plt.title('Nb of images processed per EPOCH')
plt.ylabel('Nb of images processed per epoch')
plt.xlabel('nb of GPUs')
plt.legend()
print("We achieve %s percent of scaling"% round((final_time_single/(2*final_time_mu
lti))*100,2))
```

```
We achieve 69.63 percent of scaling
```

Power Consumption comparison

7. CONCLUSION & FUTURE WORKS

Based on the results and observations from the project, we can draw several conclusions:

Performance Improvement with 2-GPU Data Parallelism: The use of 2-GPU data parallelism significantly improves the convergence times of the training phases compared to the single GPU mode. The model loss is consistently lower for the 2-GPU model right from the initial epochs, indicating faster convergence and potentially better model optimization.

Higher Accuracy with 2-GPU Model: The 2-GPU model achieves higher accuracy compared to the single-GPU model. This suggests that the additional computational resources provided by the second GPU contribute to better model training and improved performance.

Diminishing Returns in Speedup: Although the 2-GPU model demonstrates improved performance over the single-GPU model, it's important to note that the speedup is not linear. The 2-GPU model is not twice as fast as the single-GPU model due to practical factors such as propagation delay between servers. These additional factors may limit the scalability of GPU-based parallelism beyond a certain point.

Consideration of Practical Constraints: In real-world settings, factors like propagation delay between servers, communication overhead, and synchronization overhead can impact the scalability and speedup achieved by parallel computing models. It's essential to consider these practical constraints when evaluating the performance of parallel computing approaches.

Future research directions include exploring alternative parallelization strategies like model parallelism or hybrid approaches to improve performance and scalability. Optimizing communication overhead between GPUs and servers is crucial, involving techniques such as reducing data transfer sizes and optimizing network protocols. Dynamic resource allocation strategies can enhance efficiency by adaptively allocating resources based on workload characteristics. Integrating parallel computing with advanced hardware architectures, such as GPUs or TPUs, can further boost performance. Scalability studies on larger datasets are needed to understand performance under challenging conditions. Benchmarking and comparative studies with other frameworks or hardware configurations can provide insights into performance trade-offs and guide improvements in distributed training frameworks.

8. REFERENCES

1. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems by
Abadi et al. (2016)

2. Jeff Dean et al., "Large scale distributed deep networks," Advances in neural information processing systems, 2012. https://dl.acm.org/doi/10.5555/2999134.2999271

3. E. N. Watanabe and K. R. Braghetto, "Improving Parallelism in Data-Intensive

Workflows with Distributed Databases," 2018 IEEE International Conference on Services Computing (SCC), San Francisco, CA, USA, 2018, pp. 209-216, doi: 10.1109/SCC.2018.00034.
4. TensorFlow. (2023, April 18). tf.distribute.experimental.MultiWorkerMirrored Strategy.
https://www.tensorflow.org/api_docs/python/tf/distribute/MultiWorkerMirroredStrategy
5. Shallue, Christopher J., et al. ”Measuring the effects of data parallelism on neural network training.” arXiv preprint arXiv:1811.03600 (2018).
6. TensorFlow. (2023, April 18). tf.distribute
https://www.tensorflow.org/api_docs/python/tf/distribute/
7. Alexander Sergeev, Mike Del Balso: Horovod: fast and easy distributed deep learning in TensorFlow.
https://doi.org/10.48550/arXiv.1802.05799