Parallel Vision: Image Classification Using Parallelism in PyTorch and Tensorflow

Saliq Gowhar - 211AI043
Department of Information Technology
NIT Karnataka, Surathkal
saliqgowhar.211ee250@nitk.edu.in

Bhavya Sharma - 211AI011
Department of Information Technology
NIT Karnataka, Surathkal
shrmabhav.211ai011@nitk.edu.in

Ashutosh Kumar - 211AI008
Department of Information Technology
NIT Karnataka, Surathkal
ashutosh.211ai008@nitk.edu.in

***Abstract*—Deep Learning, a more advanced form of Artificial Neural Networks, faces increasing challenges in handling complex computations and large amounts of data. This is mainly because neural network structures are getting more intricate, and we're dealing with massive datasets, often called big data. Parallelism has proven to be a suitable solution to address this problem with Deep Neural Networks. In this paper, we look into the importance of parallelism in deep learning to address these challenges. We explore two ways of parallelism —Data Parallelism and Data Distributed Parallelism, while working on classifying images of handwritten digits from the MNIST dataset using TensorFlow and PyTorch. The goal is to understand how these parallel strategies can make deep learning more efficient and faster and also perform a comparative analysis between the two frameworks.**

***Index Terms*—Data Parallelism, Data Distributed Parallelism, Scaling, Tensorflow, PyTorch, Convolutional Neural Networks**

I. Introduction

Deep learning has become a widely studied field attracting researchers from diverse backgrounds due to its applications in various domains. The increasing speed of data production and its diverse origins pose a challenge that extends beyond academia, making deep learning a vibrant area of research [1]. The fundamental idea behind deep learning models is to mimic the brain's ability to visualize objects, with a key focus on capturing spatiotemporal dependencies. However, this goal is hindered by the substantial amount of data required, the need to aggregate it effectively, and the computational power necessary for efficient network training. To overcome these challenges, researchers turn to parallelization, leveraging the inherent parallelized nature of deep neural networks. As a critical facet in the deep learning landscape, parallelism and distribution have garnered significant attention, with numerous studies summarizing the state of the art in multi-core and distributed settings. Evaluating the speedup in training Convolutional Neural Networks using both single-core CPUs and GPUs [2], parallelization emerges as a pivotal solution to address the pressing challenges within deep learning, promising widespread benefits across various research domains [3].

In this paper, we delve into the implementation of a parallelized Convolutional Neural Network (CNN) for image classification, specifically focusing on handwritten digit recognition using the MNIST dataset. Our approach involves employing two distinct parallelization techniques—Data Parallelism [8] and Data Distributed Parallelism [9] —implemented using two leading deep learning frameworks, TensorFlow and PyTorch. To conduct our experiments, we leverage the computational power of the Kaggle's T4 GPU, which is equipped with two GPUs, enabling us to explore parallelism on this dual GPU setup. Our investigation includes a comprehensive analysis of crucial performance metrics such as time, accuracy, and loss, along with an exploration of the scalability of the models. Additionally, we conduct a comparative study between single GPU and multiple GPU scenarios, employing both TensorFlow and PyTorch. The paper concludes with an insightful comparison between the parallelism implementations in TensorFlow and PyTorch, shedding light on their respective strengths and weaknesses.

II. Literature Survey

Park et al. [4] employ data parallelism to create a Human Activity Recognition (HAR) application using LSTM models, containerized within a Kubernetes cluster for real-time recognition. Their study systematically assesses the impact of data parallelism on HAR application performance, specifically evaluating CPU and GPU performance in both container and host environments. Zhang et al. [5] delve into the optimization of the YOLOv4 target detection model, focusing on expediting the model training process through distributed data parallelism. In this study, the authors introduce a novel training approach incorporating cumulative gradients in a distributed data parallel setting. This method effectively addresses challenges arising from video memory limitations that hinder increasing input size. Through a series of experiments tackling batch-related issues, they empirically demonstrate the superior speedup and parallel efficiency achieved by their proposed methodology.

Sattar et al. [6] tackle the Sparse DNN Challenge, introducing a scalable solution with data parallelism on GPUs using Python TensorFlow. They utilize GraphChallenge datasets based on MNIST handwritten letters and Synthetic DNNs from RadiX-Net with varied neurons and layers. Implementing data parallelism for Sparse DNN on GPUs, their solution achieves a substantial 4.7× speedup over the baseline serial MATLAB version. Additionally, their TensorFlow GPU implementation demonstrates a 3-fold speedup compared to their own TensorFlow CPU implementation. Ganesan et al [7]. conduct a comprehensive analysis of distributed training strategies

in both synchronous and asynchronous settings. Their study involves the implementation of LSTM and MLP models on two industrial time-series datasets sourced from network management systems (NMS) and the Google cluster trace (GCT). Additionally, they apply the Inception-v3 model to the CIFAR-10 image dataset. The authors gauge the effectiveness of these strategies through empirical evaluations of key metrics, including loss, accuracy, training time, and scalability.

III. Background

This section aims to furnish a concise yet comprehensive overview of the various techniques and dataset strategically employed throughout in this work.

A. GPU Description

We have utilised the T4 GPUs from Kaggle which provide access to 2 GPUs and hence being capable for parallelism. The description of the GPUs is given in Fig 1.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.161.03   Driver Version: 470.161.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   32C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla T4            Off  | 00000000:00:05.0 Off |                    0 |
| N/A   36C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

Fig. 1. GPU Description

B. Data Description

We have used the MNIST dataset [10], a benchmark in the field of machine learning, comprises a collection of 28x28 pixel grayscale images of handwritten digits (0 through 9). Created by the National Institute of Standards and Technology (NIST), this dataset is a subset of a larger dataset originally developed for training and testing various machine learning algorithms. With 60,000 training images and 10,000 testing images, MNIST serves as a foundational dataset for tasks such as digit recognition, classification, and image processing. Each image in the dataset is associated with a corresponding label indicating the numerical digit it represents, making MNIST a widely used resource for training and evaluating image-based machine learning models.

C. Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNNs) [11] represent a powerful class of deep learning models specifically designed for visual data processing, particularly images and videos. Characterized by layers with learnable parameters, including convolutional and pooling layers, CNNs autonomously learn hierarchical features from input data. Through convolutional operations, filters scan input images to extract meaningful features, while pooling layers down-sample spatial dimensions, preserving crucial information. Fully connected layers facilitate high-level feature learning and decision-making.

D. Data Parallelism

Data Parallelism is a parallel computing technique commonly employed in deep learning, specifically for training neural networks. In this approach, the model is replicated across multiple processors or devices, and each replica processes a different subset of the training data concurrently. The updates from each replica are then aggregated to update the overall model. One way to implement Data Parallelism is through multithreading, where each thread within a single process handles a portion of the data independently. This allows for parallel processing of different batches of data, effectively reducing training time and improving overall efficiency. However, it's important to note that multithreading may encounter challenges related to the Global Interpreter Lock (GIL) in certain programming languages like Python, which can impact its scalability on multicore processors.

E. Data Distributed Parallelism

Data Distributed Parallelism is a parallel computing technique used in deep learning to train models across multiple processors or devices. In this approach, the entire model is replicated on each processing unit, and each unit is responsible for computing gradients and updating model parameters independently using its subset of the data. Unlike Data Parallelism, which typically uses multithreading within a single process, Data Distributed Parallelism relies on multiprocessing. Each processor runs a separate process with its memory space, enabling more effective parallelization by avoiding the limitations associated with the Global Interpreter Lock (GIL) in certain programming languages, such as Python. This technique is well-suited for scaling deep learning models on distributed systems, allowing for efficient processing of large datasets across multiple devices or nodes.

IV. Methodology

This section provides a detailed explanation of the methodology followed in our work in the given order.

A. Exploratory Data Analysis and Preprocessing

The Exploratory Data Analysis (EDA) of the MNIST dataset commenced with an overview of the dataset, consisting of 60,000 training images and 10,000 test images, each labeled with corresponding digits. Initial visualization showcased sample images, providing an insight into the dataset's structure. Descriptive statistics, including mean, standard deviation, and pixel value distributions, unveiled the dataset's characteristics. Box plots illustrated pixel value spread, and correlation analysis, though not standard for image data, offered a glimpse into pixel relationships. Statistical analyses, including normality testing, delved into pixel value distributions. The exploration extended to image processing techniques like edge detection, Histogram of Oriented Gradients (HOG) feature extraction, and image rotation. Data augmentation was demonstrated for enhanced dataset diversity. Average images for each digit were visualized, and dimensionality reduction techniques like PCA and T-SNE provided insights into the dataset's structure.

Lastly, image reconstruction using PCA highlighted the potential for feature reduction. The data is normalised with division by 255.

B. Model Creation

We have made use of a CNN model for our image classification task. The architecture consists of three convolutional layers with increasing filter sizes (32, 64, and 128), each followed by batch normalization, rectified linear unit (ReLU) activation, and max-pooling for spatial downsampling. The network also includes three fully connected layers with dropout for regularization (with 128, 64, and 10 neurons for classification), batch normalization, and ReLU activation. The model is compiled using sparse categorical crossentropy loss, Adam optimizer, and accuracy as the evaluation metric. This architecture aims to capture hierarchical features through convolutional layers and learn non-linear mappings for classification through densely connected layers, incorporating dropout and batch normalization for improved generalization and training stability. The complete model description is given in Fig 2.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	320
batch_normalization (BatchNormalization)	(None, 28, 28, 32)	128
re_lu (ReLU)	(None, 28, 28, 32)	0
conv2d_1 (Conv2D)	(None, 28, 28, 64)	18496
batch_normalization_1 (BatchNormalization)	(None, 28, 28, 64)	256
re_lu_1 (ReLU)	(None, 28, 28, 64)	0
max_pooling2d (MaxPooling2D)	(None, 14, 14, 64)	0
conv2d_2 (Conv2D)	(None, 14, 14, 128)	73856
batch_normalization_2 (BatchNormalization)	(None, 14, 14, 128)	512
re_lu_2 (ReLU)	(None, 14, 14, 128)	0
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 128)	0
flatten (Flatten)	(None, 6272)	0
dropout (Dropout)	(None, 6272)	0
dense (Dense)	(None, 128)	802944
batch_normalization_3 (BatchNormalization)	(None, 128)	512
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 64)	8256
batch_normalization_4 (BatchNormalization)	(None, 64)	256
dropout_2 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 10)	650

Fig. 2. CNN Model Description

C. Data Parallelism in Tensorflow

In this study, we employed a data parallelism approach using TensorFlow to enhance the training efficiency of the CNN. The dataset was divided across multiple GPUs, and the batch size was dynamically adjusted based on the number of utilized GPUs (2) which resulted in a batch size of 2048. The TensorFlow (tf.distribute.Strategy API) was leveraged to enable synchronous training on multiple devices, ensuring model consistency with the use of All-reduce algorithm. A multi-GPU model was constructed and trained using the fit function with the distributed dataset, while a single-GPU model was also trained for performance comparison. The experimental setup involved caching, shuffling, and prefetching the training data to optimize data pipeline performance. The entire process, from model construction to training, was systematically timed to quantify the training acceleration achieved through data parallelism. The study encompassed 30 training epochs for both multi and single GPU training,while calculating the time taken per epoch, accuracy and loss per epoch and total time of execution.

D. Data Parallelism in PyTorch

In this study, a data parallelism approach was employed using PyTorch to enhance the training efficiency of the CNN. The dataset was preprocessed using torchvision transforms, with images converted to tensors and normalized. For training, a batch size of 2048 was utilized and for testing a batch size of 100 was used, and the MNIST dataset was divided into batches using PyTorch DataLoader. The CNN model underwent 30 epochs of training using stochastic gradient descent as the optimizer. Throughout each epoch, the model processed the training data in parallel on the available GPUs, with the optimizer updating the model parameters based on computed gradients. Training progress, including the loss at every 100 steps, was monitored for each epoch. Concurrently, the model's accuracy was evaluated on a separate test set after each epoch, providing insights into its performance. The entire training process was systematically timed to measure the duration for each epoch, and the study aimed to quantify the impact of data parallelism on training efficiency, considering factors such as accuracy and overall training time. Same process was tested on a single GPU as well for comparative analysis.

E. Data Distributed Parallelism in PyTorch

In this survey, we implemented data distributed parallelism using PyTorch to enhance the training efficiency of a CNN. The experiment involved the utilization of a single node, equipped with 2 GPUs, and the training process was orchestrated through PyTorch's distributed training capabilities. The torch.multiprocessing.spawn function facilitated the parallel execution of the training function on each GPU, with the number of nodes, GPUs per node, and total training epochs

provided as user-defined parameters through an argument parser from command line. The communication between these processes was established using the NCCL backend with master environment as Local Host and master port 5555, and each process was assigned a unique rank to enable coordinated updates during training. The model was replicated across GPUs using PyTorch's nn.parallel.DistributedDataParallel. Key training parameters included a batch size of 2048, the Adam optimizer with a learning rate of 1e-3, and the CrossEntropy loss function. The training dataset distribution across processes was managed by PyTorch's DistributedSampler to synchronize updates using All-round algorithm. Metrics such as loss and accuracy were tracked for each epoch, providing a comprehensive evaluation of the model's performance.

V. Results and Comparative Analysis

We evaluated the performance using various metrics including Time per epoch, total time of training, accuracy, loss, classification report and scaling percentage. The results for each experiment are given below.

A. Data Parallelism in Tensorflow

For 2 GPUs:

- Average Time per Epoch: 3 sec 105 ms
- Total time taken: 104.56 seconds
- Test Accuracy: 99.53%
- Scaling: 73.62%

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	980
1	0.99	1.00	1.00	1135
2	1.00	0.99	1.00	1032
3	0.99	1.00	1.00	1010
4	0.99	1.00	0.99	982
5	1.00	0.99	0.99	892
6	1.00	0.99	1.00	958
7	0.99	0.99	0.99	1028
8	1.00	1.00	1.00	974
9	0.99	0.99	0.99	1009
accuracy			1.00	10000
macro avg	1.00	1.00	1.00	10000
weighted avg	1.00	1.00	1.00	10000

Fig. 3. Classification Report for 2 GPUS- Data Parallelism Tensorflow

For 1 Gpu:

- Average Time per Epoch: 5 sec 160 ms
- Total time taken: 153.95 seconds
- Test Accuracy: 99.42%

Overall Comparison: Data Parallelism has proved to be more efficient than single GPU training in terms of all the metrics used in our experimentations.

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	980
1	0.99	1.00	1.00	1135
2	1.00	0.99	1.00	1032
3	0.99	1.00	1.00	1010
4	0.99	1.00	0.99	982
5	1.00	0.99	0.99	892
6	1.00	0.99	1.00	958
7	0.99	0.99	0.99	1028
8	1.00	1.00	1.00	974
9	0.99	0.99	0.99	1009
accuracy			1.00	10000
macro avg	1.00	1.00	1.00	10000
weighted avg	1.00	1.00	1.00	10000

Fig. 4. Classification Report for 1 GPU- Data Parallelism Tensorflow

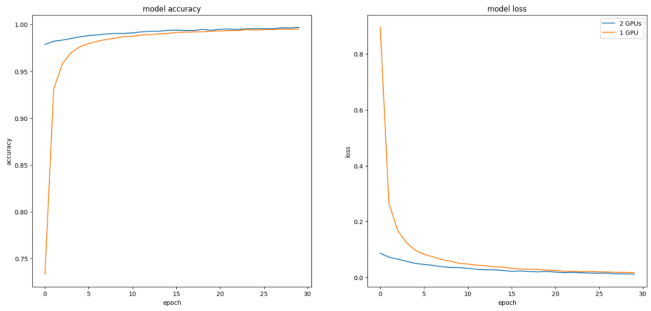

Fig. 5. Accuracy and Loss - Tensorflow

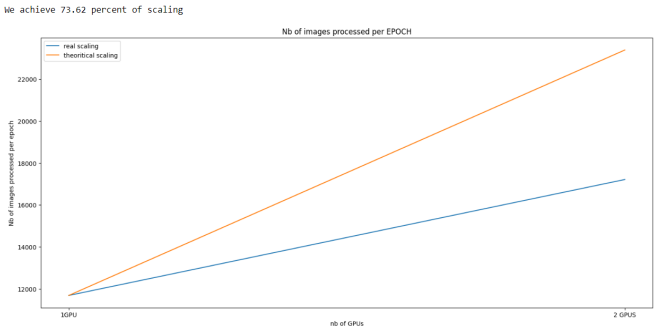

Fig. 6. Scaling in Tensorflow

B. Parallelism in PyTorch

Single GPU:

- Average Time per Epoch: 13 sec 110 ms
- Total time taken: 390.95 seconds
- Test Accuracy: 99.10%

Data Parallelism- 2 GPUs:

- Average Time per Epoch: 13 sec 110 ms
- Total time taken: 398.5 seconds
- Test Accuracy: 99.52%
- Scaling: 48.5%

Data Distributed Parallelism - 2 GPUs:

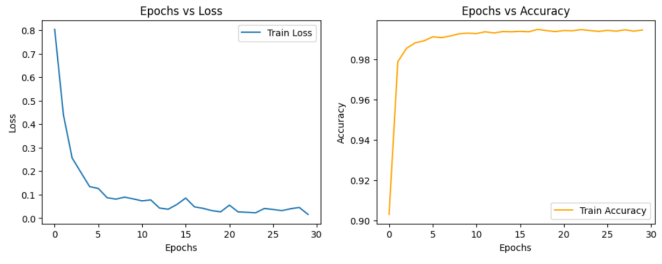

Fig. 7. Performance on 1 GPU- PyTorch

Classification Report:	precision	recall	f1-score	support
0	1.00	1.00	1.00	980
1	1.00	1.00	1.00	1135
2	1.00	0.99	1.00	1032
3	0.99	1.00	1.00	1010
4	0.99	0.99	0.99	982
5	1.00	0.99	0.99	892
6	1.00	0.99	0.99	958
7	0.99	1.00	0.99	1028
8	0.99	1.00	1.00	974
9	0.99	0.99	0.99	1009
accuracy			0.99	10000
macro avg	0.99	0.99	0.99	10000
weighted avg	0.99	0.99	0.99	10000

Fig. 8. Classification Report on single GPU- PyTorch

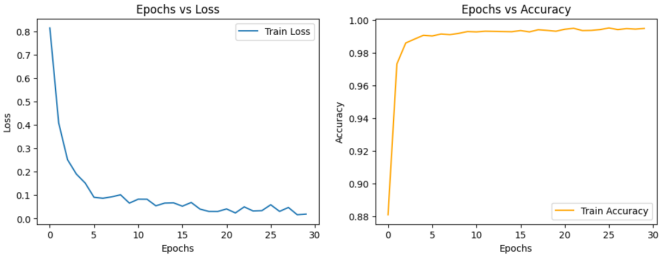

Fig. 9. Data Parallelism Performance on 2 GPUs- PyTorch

- Average Time per Epoch per GPU: 3 sec 500 ms
- Total time taken: 110.96 seconds
- Test Accuracy: 99.8%
- Scaling: 184.88%

Overall Remarks: Data Parallelism does not show improvement with respect to Single GPU training in PyTorch due to the fact that it is based on multi-threading , hence prone to GIL and mutex locks, whereas Data Distributed Parallelism shows significant impovement with respect to both Single GPU Training as well as Data Parallelism due to the fact that it is based on multi-processing and model is replicated only once in entire lifetime.

VI. Conclusion and Future Wroks

In conclusion, our exploration into the parallelized implementation of Convolutional Neural Networks (CNNs) for handwritten digit recognition on the MNIST dataset using TensorFlow and PyTorch has yielded valuable insights. We've dissected the intricacies of data parallelism and data distributed parallelism, assessing their impact on key performance metrics such as time efficiency, accuracy, and loss. We conclude that parallelism is indeed an advancement towards faster computation in Deep Learning. Additionally, we conclude that Data Parallelism in Tensorflow is superior to Pytorch due to high scaling, and that Distributed Data Parallelism has an upper edge over Data Parallelism due to use of Multi-Processing

Classification Report:	precision	recall	f1-score	support
0	0.99	1.00	1.00	980
1	1.00	1.00	1.00	1135
2	0.99	1.00	1.00	1032
3	0.99	1.00	1.00	1010
4	1.00	0.99	0.99	982
5	1.00	0.99	0.99	892
6	1.00	0.99	1.00	958
7	1.00	0.99	0.99	1028
8	0.99	0.99	0.99	974
9	0.99	0.99	0.99	1009
accuracy			0.99	10000
macro avg	0.99	0.99	0.99	10000
weighted avg	0.99	0.99	0.99	10000

Fig. 10. Classification Report using Data Parallelism on 2 GPUs- PyTorch

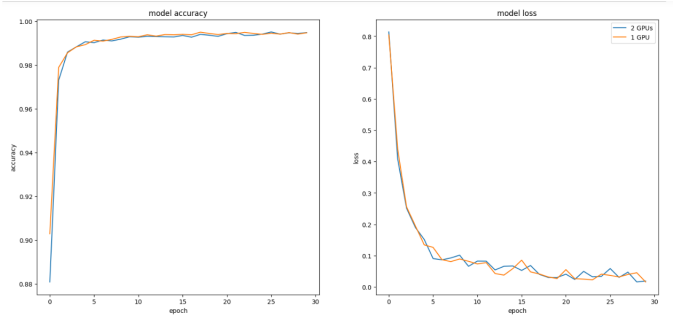

Fig. 11. Data Parallelism Performance Comparison with 1 GPU Training- PyTorch

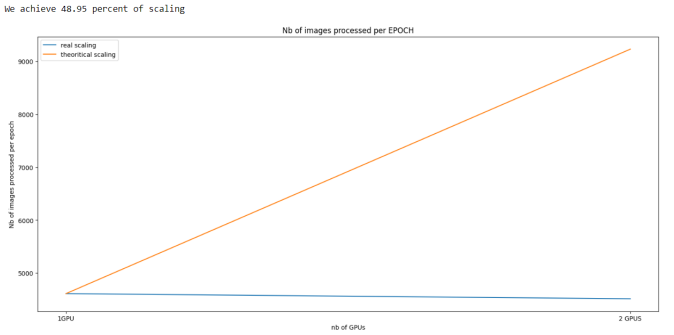

Fig. 12. Scaling using Data Parallelism - PyTorch

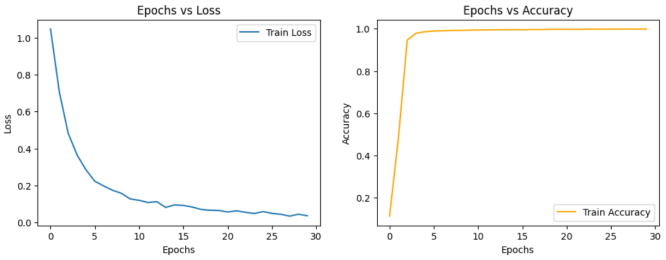

Fig. 13. Data Distributed Parallelism Performance on 2 GPUs- PyTorch

Classification Report:

	precision	recall	f1-score	support
0	0.99	1.00	1.00	509
1	0.99	1.00	1.00	571
2	0.99	1.00	1.00	512
3	0.99	0.99	0.99	533
4	1.00	0.98	0.99	505
5	0.99	1.00	0.99	441
6	1.00	0.99	0.99	480
7	1.00	0.99	0.99	504
8	0.99	0.99	0.99	462
9	0.98	0.99	0.98	483
accuracy			0.99	5000
macro avg	0.99	0.99	0.99	5000
weighted avg	0.99	0.99	0.99	5000

Fig. 14. Classification Report using Data Distributed Parallelism on 2 GPUs- PyTorch

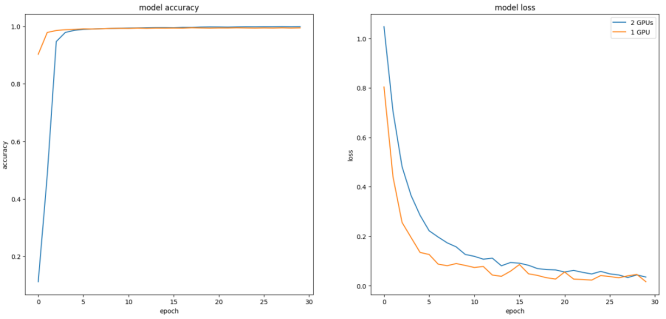

Fig. 15. Data Distributed Parallelism Performance Comparison with 1 GPU Training- PyTorch

rather than Multi-Threading, hence avoiding Mutex Locks and yielding better results.

Moving forward, there are promising avenues for future research. Firstly, exploring more diverse datasets beyond MNIST could provide a deeper understanding of how these parallelization techniques generalize across different domains. Additionally, investigating advanced parallelization strategies like model parallelism and optimizations, as well as delving into real-world applications beyond image classification, could further enhance the practical utility of parallelized CNNs. Finally, addressing the interpretability and explainability of models trained with parallelization techniques remains an important area for future exploration, ensuring the trustworthiness of these advanced learning systems in real-world scenarios.

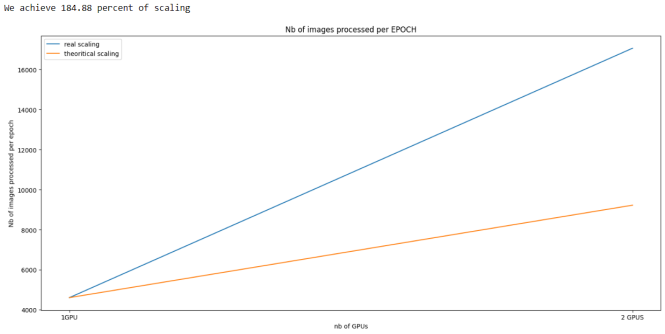

Fig. 16. Scaling using Data Distributed Parallelism - PyTorch

REFERENCES

[1] V. Giansanti, S. Beretta, D. Cesini, D. D'Agostino and I. Merelli, "Parallel Computing in Deep Learning: Bioinformatics Case Studiesa," 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Pavia, Italy, 2019, pp. 329-333, doi: 10.1109/EMPDP.2019.8671556.

[2] V. Hegde and S. Usmani, "Parallel and Distributed Deep Learning" in Tech. report Stanford University, June 2016.

[3] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald and E. Muharemagic, "Deep learning applications and challenges in big data analytics", Journal of Big Data, vol. 2, no. 1, 2015.

[4] T. D. T. Nguyen et al., "Performance Analysis of Data Parallelism Technique in Machine Learning for Human Activity Recognition Using LSTM," 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Sydney, NSW, Australia, 2019, pp. 387-391, doi: 10.1109/CloudCom.2019.00066.

[5] X. Zhang, C. Zhang and M. Jiang, "Distributed Data Parallel Training Based on Cumulative Gradient," 2022 2nd International Conference on Computer, Control and Robotics (ICCCR), Shanghai, China, 2022, pp. 202-206, doi: 10.1109/ICCCR54399.2022.9790196.

[6] N. S. Sattar and S. Anfuzzaman, "Data Parallel Large Sparse Deep Neural Network on GPU," 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), New Orleans, LA, USA, 2020, pp. 1-9, doi: 10.1109/IPDPSW50202.2020.00170.

[7] G. Ponnuswami, S. Kailasam and D. A. Dinesh, "Evaluating Data-Parallel Distributed Training Strategies," 2022 14th International Conference on COMmunication Systems NETworkS (COMSNETS), Bangalore, India, 2022, pp. 759-763, doi: 10.1109/COMSNETS53615.2022.9668349.

[8] B. Shinde and S. T. Singh, "Data parallelism for distributed streaming applications," 2016 International Conference on Computing Communication Control and automation (ICCUBEA), Pune, India, 2016, pp. 1-4, doi: 10.1109/ICCUBEA.2016.7859983.

[9] M. Hsu, "Parallel computing with distributed shared data," [1989] Proceedings. Fifth International Conference on Data Engineering, Los Angeles, CA, USA, 1989, pp. 485-, doi: 10.1109/ICDE.1989.47253.

[10] L. Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]," in IEEE Signal Processing Magazine, vol. 29, no. 6, pp. 141-142, Nov. 2012, doi: 10.1109/MSP.2012.2211477.

[11] S. Albawi, T. A. Mohammed and S. Al-Zawi, "Understanding of a convolutional neural network," 2017 International Conference on Engineering and Technology (ICET), Antalya, Turkey, 2017, pp. 1-6, doi: 10.1109/ICEngTechnol.2017.8308186.