Exploring Various Configurations of Distributed Training on CIFAR-10 For Classification

April 29, 2025

Abstract

Deep learning models require substantial computational resources for effective training, making distributed training essential for scalability and efficiency. This project investigates the impact of different distributed training configurations on accuracy and training time for image classification using the CIFAR-10 dataset. We compare Single Node (1 GPU), Single Node (2 GPUs), and Multi-Node (2 GPUs each) setups under various mini-batch configurations. Utilizing PyTorch's Distributed Data Parallel (DDP) framework, we implement an optimized training pipeline to analyze how different hyperparameter choices affect model performance. Our experiments, using three distinct architectures, offer a comprehensive evaluation of distributed training strategies. **Project Link: https://github.com/vivekdhir77/Distributed-Training-CIFAR10**

1 Introduction

Training deep neural networks is both computationally intensive and resource-demanding. Distributed training mitigates these challenges by leveraging multiple GPUs to parallelize computations, thereby reducing training time. This project explores the trade-offs between training time and accuracy across various configurations, analyzing the effects of key hyperparameters, such as mini-batch size. Here we focus on image classification, a fundamental problem in computer vision that plays a crucial role in industries handling large-scale image data. We utilize the CIFAR-10 dataset, which consists of 60,000 32x32 color images across 10 classes, as a benchmark to evaluate the efficacy of distributed training approaches.

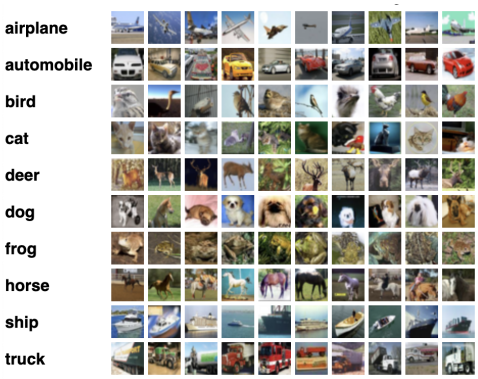


Figure 1: Sample images from the CIFAR-10 dataset, illustrating 10 distinct classes.

1.1 Related Work

Shallue et al. (2018) [1] researched the impact of data parallelism on training neural networks, detailing how it would reduce training time by distributing the computations across several GPUs. Patwardhan et al. (2024) [2] even contrasted distributed approaches with training GPT-2 and debated how model accuracy is compromised by training efficiency.

Model parallelism is also important in distributed systems. Choi et al. (2023) [3] discussed improving model parallelism in distributed deep learning, whereas Brakel et al. (2024) [4] provided a comprehensive

survey of model parallelism, particularly for large-scale systems. Archer et al. (2023) [5] proposed methods such as pipelined inference to improve the efficiency of distributed systems, achieving maximum throughput during training and inference.

Practical manuals for distributed training deployment are found in articles like Boehm (2022) [6] and Ee (2022) [7], who described PyTorch's Distributed Data Parallel (DDP) system and showed how to scale training with multiple GPUs efficiently.

Building upon these foundations, our work investigates the practical implementation of distributed training using PyTorch's DDP framework with the NCCL backend. We employ data parallelism where each GPU processes a subset of the mini-batch, with gradients synchronized using the All-Reduce algorithm. Our experiments explore three key configurations: Single Node (1 GPU), Single Node (2 GPUs), and Multi-Node (2 GPUs), while maintaining a recommended bucket size of 25 MiB for optimal computation communication overlap. This approach allows us to systematically analyze the trade-offs between training efficiency and model accuracy across different architectures.

2 Technical Approach

2.1 Architecture 1: Simple CNN

The ConvNet defined for a simple CNN, has two convolutional layers followed by two fully connected layers. The first convolutional layer (conv1) takes a 3-channel RGB image and applies 4 filters of size 3×3, followed by a 2×2 max pooling operation. The second layer (conv2) uses 8 filters of size 3×3, again followed by a 2×2 max pooling. After the convolution and pooling steps, the feature maps are flattened and passed through a fully connected layer with 32 neurons. Finally, the output layer maps to 10 classes, CIFAR-10 dataset. The model is lightweight and efficient, trading depth and complexity for faster training and simplicity, ideal for experimentation and low-resource scenarios.

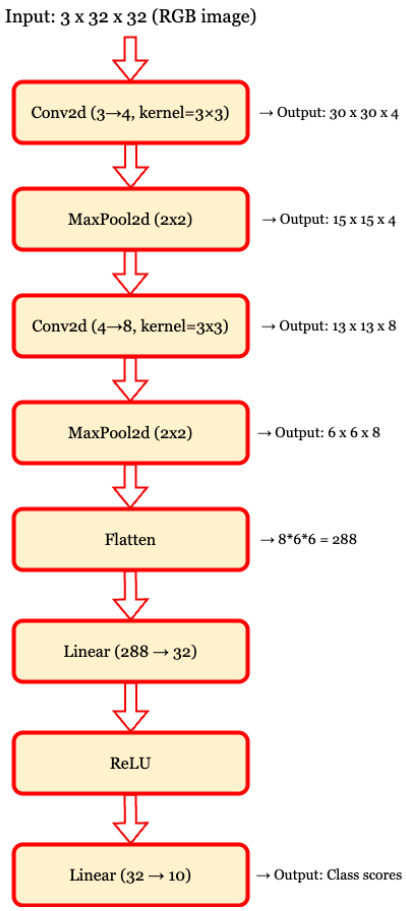


Figure 2: Simple CNN

2.2 Architecture 2: Deep CNN

We defined a configurable ResNet (Residual Network) architecture for our deep CNN, commonly used for image classification tasks. It is built using residual blocks (BasicBlock) to allow gradients to flow through the network more effectively, solving the vanishing gradient problem in deep networks. The ResNet uses the BasicBlock structure, which includes two 3×3 convolutions and a skip connection (identity or 1×1 convolution when dimensions differ).

The ResNet starts with a 3×3 convolution and batch norm layer, followed by four stages of residual blocks with increasing filter sizes and downsampling (stride=2). After the final stage, it applies global average pooling and a fully connected layer to produce class scores for the 10 classes.

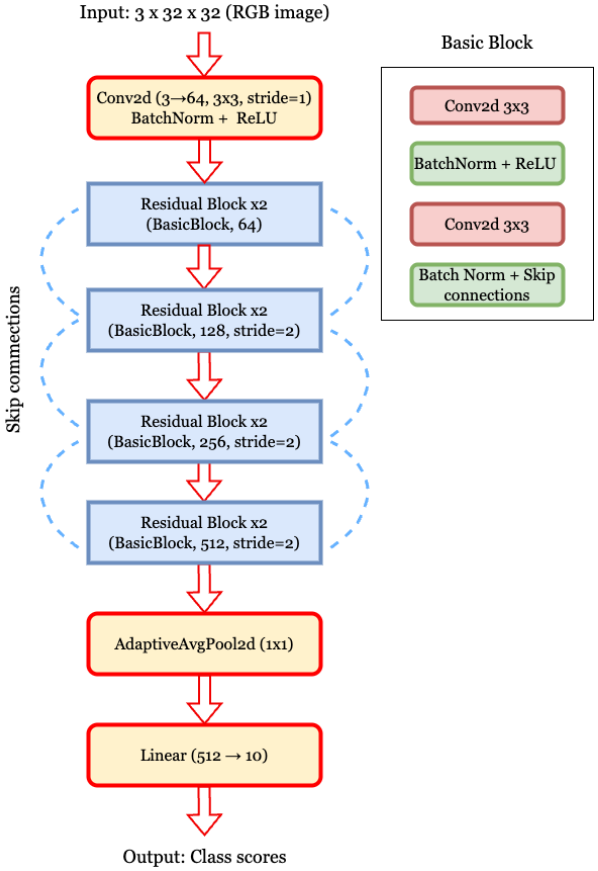


Figure 3: Deep CNN

2.3 Architecture 3: Vision Transformer

A Vision Transformer (ViT) model, does image classification by using transformer architectures originally developed for NLP tasks. Instead of using convolutional layers throughout, ViT starts by dividing the input image into non-overlapping patches using the EmbedLayer, where each patch is linearly projected into an embedding space. A learnable [CLS] token is prepended, and positional encodings are added to retain spatial information. The embedded sequence then passes through a stack of TransformerBlocks. Each block uses multi-head self-attention (MultiHeadSelfAttention) to model relationships across all patches globally, followed by a feed-forward network with GELU activations and residual connections enhanced with dropout and optional stochastic depth for regularization. Finally, the output corresponding to the [CLS] token is passed through a classifier to produce the final class prediction.

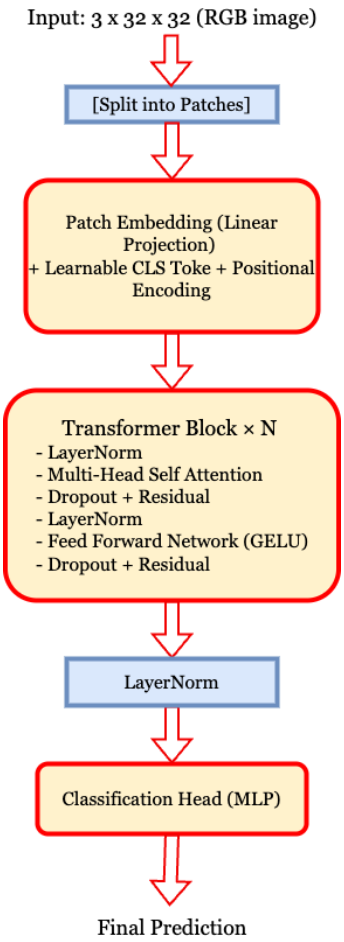


Figure 4: ViT

2.4 Distributed Training Strategy

Our Distributed training is built using PyTorch's Distributed Data Parallel (DDP) framework, using the NCCL backend for efficient GPU communication. Key components include:

- Model initialization and synchronization across processes
- Distributed data loading with `DistributedSampler`
- Gradient synchronization using DDP's All-Reduce mechanism
- Model checkpointing for fault tolerance

2.4.1 Data Parallelism Strategy

We used data parallelism, where each GPU handles a unique subset of the mini-batch. The model is replicated across devices, and during the backward pass, gradients from each GPU are aggregated using All-Reduce. Our experiments were conducted using NVIDIA Quadro P4000 GPUs, which feature the Pascal architecture with 1792 CUDA cores and 8GB GDDR5 memory per card. On which We explored three GPU configurations:

- **Single Node (1 GPU):** One Quadro P4000 with 8GB GDDR5 memory and 256-bit memory interface, capable of up to 243 GB/s memory bandwidth
- **Single Node (2 GPUs):** Two Quadro P4000s in a single system, providing a combined 16GB GDDR5 memory and 3584 CUDA cores
- **Multi-Node Setup:** two nodes, each equipped with two Quadro P4000s, connected via a private network

The distributed setup utilizes 250GB shared storage system accessible across all nodes for consistent data access, private network infrastructure for inter-node communication and NCCL backend for GPU communication, leveraging the P4000's PCIe 3.0 x16 interface

2.4.2 Synchronization and Gradient Communication in NCCL

Gradient communication and synchronization in our DDP setup are managed using bucket algorithm. Gradients are grouped into buckets and communicated once a bucket is filled. For this project:

- We use a bucket size of 25 MiB, as recommended for typical workloads.
- Smaller bucket sizes offer finer-grained synchronization, reducing memory usage but increasing communication overhead.
- Larger bucket sizes reduce communication overhead but can delay updates and increase memory usage.

2.4.3 Experiment Automation and Logging

Each experiment is automatically logged with:

- Accuracy, precision, recall, f1-score and specificity scores.
- Training time and per-epoch duration.
- Checkpoints for fault-tolerant recovery.

3 Experimental Results

3.1 Dataset

The CIFAR-10 dataset (Figure 1) consists of 60,000 images across 10 mutually exclusive classes, with 50,000 images allocated for training and 10,000 for testing.

The ten classes in CIFAR-10 include: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck, with no overlap between categories.

3.2 Outcomes

3.2.1 Simple CNN

From Table 1, we see time taken for training reduces by nearly 33% for single node GPU configuration and around 21% for multi-node GPU configuration by batch sizes. And from single node to multi-node, there is a decrease of at least 54% in training time.

Table 1: Training Time (in seconds) for Simple CNN

GPU Configuration	Batch Size 16	Batch Size 64
Single Node (1 GPU)	393.81	260.85
Single Node (2 GPUs)	231.96	156.68
Multi-Node (2 GPUs)	90.79	70.90

From Table 2, larger batch sizes (64) led to considerable performance decrease compared to smaller batch sizes (16), with most metrics declining by at least 50%. Only specificity remained relatively stable with a minimal 2% decrease.

When comparing configuration setups, the multi-node configuration performed worse than single-node setups. Accuracy decreased by at least 21%, while precision and F1-score dropped by at least 50%. Recall declined by at least 23%, and even specificity showed a small 1% reduction.

Table 2: Metrics for Simple CNN

GPU Configuration	Accuracy		Precision		Recall		F1-Score		Specificity	
	BS=16	BS=64	BS=16	BS=64	BS=16	BS=64	BS=16	BS=64	BS=16	BS=64
Single Node (1 GPU)	0.3981	0.1458	0.39407	0.1502	0.3981	0.1458	0.3934	0.0840	0.93313	0.9051
Single Node (2 GPUs)	0.3341	0.1476	0.31828	0.1629	0.3341	0.1714	0.31035	0.1178	0.92602	0.9079
Multi-Node (2 GPUs)	0.2448	0.1138	0.1958	0.0422	0.2448	0.1114	0.1946	0.0333	0.9161	0.9013

3.2.2 Deep CNN

From Table 3, we observe that training time reduces significantly as we move from smaller to larger batch sizes. For single node configurations, increasing batch size from 16 to 64 reduces training time by approximately 30%. The multi-node configuration shows a reduction of around 20% when increasing the batch size.

With different GPU configurations, the performance improved. From a single node with one GPU to a single node with two GPUs, it resulted in training time reductions of approximately 46% for both batch sizes. The most dramatic improvement occurs when comparing the multi-node configuration to the single node with one GPU, where training times decrease by nearly 75% for both batches. Even when compared to the single node with two GPUs setup, the multi-node configuration delivers significant speedups of approximately 50% for batches.

Table 3: Training Time (in seconds) for Deep CNN

GPU Configuration	Batch Size 16	Batch Size 64
Single Node (1 GPU)	1185.90	819.44
Single Node (2 GPUs)	634.60	448.42
Multi-Node (2 GPUs)	298.57	237.82

From table 4, for batch size comparisons, increasing from 16 to 64 gives in slight performance improvement across all configurations: Single Node (1 GPU) metrics improved by approximately 0.6%, Single Node (2 GPUs) by 1.3%, and Multi-Node (2 GPUs) by 0.7%. Specificity remained extremely stable with minimal changes of about 0.1% across all setups.

When comparing GPU configurations, both distributed setups showed performance decreases compared to Single Node (1 GPU). The Single Node (2 GPUs) experienced the largest decline, with metrics(accuracy, precision, recall and f1-score) decreasing by approximately 6.2% for both batch sizes. The Multi-Node (2 GPUs) configuration performed slightly better, with metrics(accuracy, precision, recall and f1-score) decreasing by about 3.9% across both batch sizes. The specificity remained stable across all configurations. Overall, the Deep CNN shows more consistent performance across configurations than the Simple CNN, with the single GPU setup yielding the best results and larger batch sizes providing slight improvements.

Table 4: Metrics for Deep CNN

GPU Configuration	Accuracy		Precision		Recall		F1-Score		Specificity	
	BS=16	BS=64	BS=16	BS=64	BS=16	BS=64	BS=16	BS=64	BS=16	BS=64
Single Node (1 GPU)	0.8496	0.8548	0.8498	0.8542	0.8496	0.8548	0.8490	0.8542	0.9833	0.9839
Single Node (2 GPUs)	0.7946	0.8047	0.7942	0.8048	0.7946	0.8047	0.7935	0.8044	0.9772	0.9783
Multi-Node (2 GPUs)	0.8164	0.8215	0.8157	0.8219	0.8164	0.8215	0.8156	0.8215	0.9796	0.9802

3.2.3 Vision Transformer

From table 5, increasing the batch size from 16 to 64 reduced training time by approximately 71% across all configurations.

When comparing different GPU configurations, single node (2 GPU) reduces training time by approximately 45% for both batch sizes. The multi-node configuration reduces training time by about 62% compared to a single GPU setup. When compared to the single node with two GPUs, the multi-node configuration still shows significant improvements of around 30-31%. These results show that ViT models can scale efficiently across distributed computing environments.

Table 5: Training Time (in seconds) for ViT

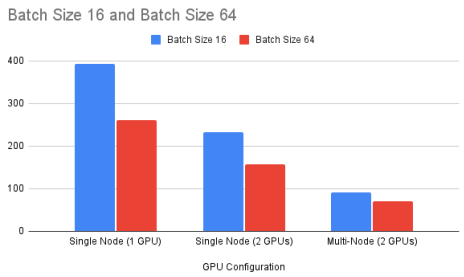
GPU Configuration	Batch Size 16	Batch Size 64
Single Node (1 GPU)	1386.59	397.59
Single Node (2 GPUs)	752.56	217.38
Multi-Node (2 GPUs)	525.28	149.95

From table 6, for the Single Node (1 GPU) setup, increasing batch size from 16 to 64 yields moderate performance improvements across all metrics (1.1-1.3%). However, for distributed configurations: Single Node (2 GPUs) shows slight decreases (0.7-1.3%), while Multi-Node (2 GPUs) shows more significant performance drops (5-6.6%).

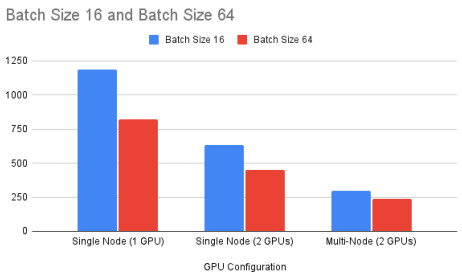
When comparing GPU configurations with batch size 16, both distributed setups show minimal performance decreases in accuracy, precision, recall and f1-score when compared to Single Node (1 GPU). However, with batch size 64, Single Node (2 GPUs) shows moderate decreases (2.6-3.5%), while Multi-Node (2 GPUs) experiences performance decrease of 6.8-7.6% in accuracy, precision, recall and f1-score. Specificity remains stable across all configurations, with minimal changes (0.1-0.6%).

Table 6: Metrics for ViT

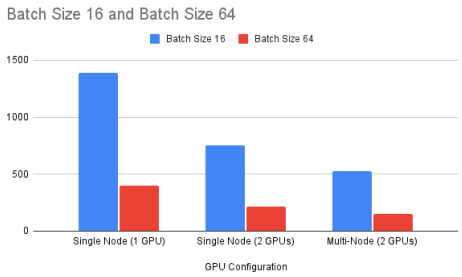
GPU Configuration	Accuracy		Precision		Recall		F1-Score		Specificity	
	BS=16	BS=64	BS=16	BS=64	BS=16	BS=64	BS=16	BS=64	BS=16	BS=64
Single Node (1 GPU)	0.6691	0.6775	0.6736	0.6811	0.6691	0.6775	0.6588	0.6662	0.9632	0.9642
Single Node (2 GPUs)	0.6627	0.6539	0.6720	0.6635	0.6627	0.6539	0.6511	0.6466	0.9625	0.9615
Multi-Node (2 GPUs)	0.6621	0.6271	0.6737	0.6291	0.6621	0.6271	0.6534	0.6206	0.9625	0.9586



(a) Training times for Simple CNN

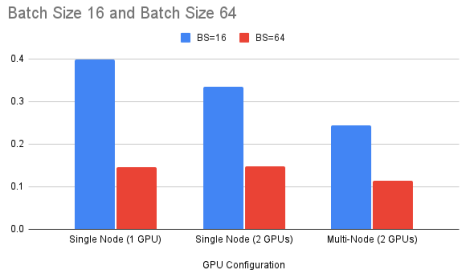


(b) Training times for Deep CNN

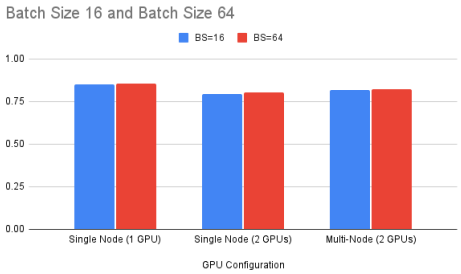


(c) Training times for ViT

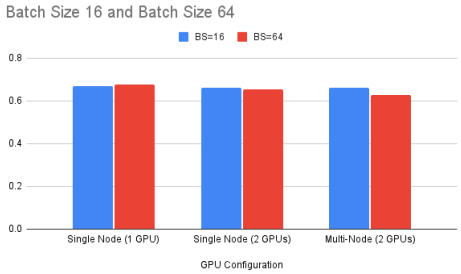
Figure 5: Comparison of training times across different models.



(a) Accuracy for Simple CNN



(b) Accuracy for Deep CNN



(c) Accuracy for ViT

Figure 6: Comparison of Accuracy times across different models and different setups.

4 Conclusions and Future Work

Our evaluation of Simple CNN, Deep CNN, and Vision Transformer architectures across different GPU configurations showed distinct performance patterns. Deep CNN gave superior performance (accuracy ~ 0.8) compared to ViT (accuracy ~ 0.67) and Simple CNN (accuracy $\sim 0.14 - 0.39$).

The three architectures responded differently to batch size changes. Simple CNN showed substantial degradation in all metrics ($> 50\%$) with larger batches, while Deep CNN showed slight improvements (0.6-1.3%) and ViT showed mixed results depending on configuration. More complex architectures showed greater resilience to distributed computing challenges.

All models experienced performance decreases in multi-node setups, with Simple CNN showing the largest degradation in all metrics(21-50%), followed by ViT (1-7%) and Deep CNN (3-6%). However, training times improved significantly with distributed computing, especially for ViT (71% reduction with larger batches, 45-62% with multiple GPUs).

For the future, our project can extend and focus on architecture-specific scaling strategies, where we can develop optimized scaling approaches tailored to specific architectures to help maintain performance while leveraging distributed computing benefits. By improving communication protocols for multi-node set ups can potentially reduce the performance degradation seen. Additionally, we can test these configurations on more complex models and larger datasets to determine if the observed patterns hold at greater scales.

References

[1] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, "Measuring the effects of data parallelism on neural network training," *arXiv preprint arXiv:1811.03600*, 2018.

[2] I. Patwardhan, S. Gandhi, O. Khare, A. Joshi, and S. Sawant, "A comparative analysis of distributed training strategies for gpt-2," *arXiv preprint arXiv:2405.15628*, 2024.

[3] H. Choi, B. H. Lee, S. Y. Chun, and J. Lee, "Towards accelerating model parallelism in distributed deep learning systems," *PLOS ONE*, vol. 18, no. 11, p. e0293338, 2023.

[4] F. Brakel, U. Odyurt, and A.-L. Varbanescu, "Model parallelism on distributed infrastructure: A literature review from theory to llm case-studies," *arXiv preprint arXiv:2403.03699*, 2024.

[5] A. Archer, M. Fahrbach, K. Liu, and P. Prabhu, "Practical performance guarantees for pipelined dnn inference," *arXiv preprint arXiv:2311.03703*, 2023.

[6] S. Boehm, "Data parallel training." `https://siboehm.com/articles/22/data-parallel-training`, 2022. Accessed: February 27, 2025.

[7] Y. Ee, "Pytorch distributed: A bottom-up perspective." `https://medium.com/@eeyuhao/pytorch-distributed-a-bottom-up-perspective-e3159ee2c2e7`, 2022. Accessed: February 27, 2025.