Distributed Learning in Deep Neural Networks

Zurehma Ayesha Rameez
Politecnico di Torino
Turin, Italy
s323502@polito.studenti.it

Nicolai H. Faye
Politecnico di Torino
Trondheim, Norway
s335057@polito.studenti.it

Abstract

As modern AI models increase in both complexity and scale, distributed training is needed to handle larger datasets and architecture. This paper presents an overview of modern distributed learning strategies, and how we potentially can optimize these. In our experiments we used the LeNet-5 model on the CIFAR-100 dataset. During the initial phase of our experiments we ran a centralized baseline with the best hyperparameters for different optimizers to give us a reference. We also explored modern large batch optimizers such as LARS and LAMB with increasing batch sizes. Furthermore we implemented LocalSGD and experimented with different combinations of workers and local steps to find the optimal combination for distributed learning. Our findings provide practical insights into the optimization of distributed training setups, focusing on batch sizes, optimizer selection, and configurations for scalable and efficient deep learning. Our code is available on Github

1. Introduction

The field of deep learning has experienced unprecedented growth, with neural networks becoming increasingly more sophisticated and capable of solving complex tasks across diverse domains. However, as models grow in complexity with huge datasets, the computational demands of training these models have pushed the boundaries of what single-machine setups can achieve. This has led to a shift towards distributed training paradigms, where the workload is distributed over multiple devices or nodes. While this transition sounds promising because of the reduced training times and the ability to handle larger models, it introduces fundamental challenges that expose limitations of traditional optimizers such as Stochastic Gradient Descent (SGD) and its derivatives when applied in distributed settings.

The primary challenge lies in the communication overhead and synchronization requirements inherent in distributed training. When SGD is naively extended to distributed environments, frequent gradient synchronization across nodes can create bottlenecks that diminish the benefits of parallelization. The increased batch sizes present in distributed settings can also lead to reduced performances. To overcome these limitations, specialized optimizers such as 'Layer-wise Adaptive Rate Scaling' (LARS) and 'Layer-wise Adaptive Moments optimizer for Batch training' (LAMB) address the challenges by adapting learning rates on a per-layer basis, improving stability and convergence when gradients vary significantly across different layers. Despite these advances, large batch training often reduces generalization performance. Therefore, we explored methods such as LocalSGD which balances communication overhead by allowing workers to perform multiple local updates before synchronization.

For all experiments we used a variation of the LeNet-5 model on the CIFAR-100. We used a train-test ratio of 5:1 of the dataset. We pre-processed the training set with common data augmenation techniques such as random cropping and horizontal flipping as well as normalization (both train and test sets) to minimize the generalization gap. The schedulers used in our experiments were the cosine annealing scheduler and the step scheduler to adjust learning rates dynamically allowing faster convergence and improved performance.

2. Related work

Training large machine learning models efficiently across multiple machines is essential, but it comes with challenges. Using very large batch sizes can help, but only if the training settings are carefully tuned; otherwise, performance can suffer.

To deal with these issues, advanced optimizers like LARS [2] and LAMB [3] have been developed. LARS adjusts the learning rate for each layer individually, making training more stable. LAMB goes further by normalizing updates both across individual parameters and entire layers, which helps models converge better, especially on complex tasks.

Another approach is LocalSGD, which improves on standard mini-batch SGD by reducing communication between workers. It often results in lower training loss and higher test accuracy. [1]

3. Centralized Learning

In centralized learning all training data and computation is handled on a single machine or tightly coupled system. The unified environment ensures complete control over the training process, and it does not introduce any communication overhead as all the data and computations remain within the same system. Some of the most prominent optimization algorithms include SGD and its variations. This section aims to explain the behaviour of this optimizer in our centralized setting, leveraging the Pytorch framework.

3.1. SGDM

Stochastic Gradient Descent Momentum (SGDM) is a variation of the traditional SGD, which includes a momentum term that accelerates convergence and increases stability during training. The momentum mechanism accumulates an average of past gradients allowing the convergence to accelerate and dampen oscillations during training. The SGDM optimizer was implemented using the PyTorch torch.optim.SGD class, setting the momentum coefficient as a hyperparameter.

4. Large Batch Optimizers

Training neural networks with very large batch sizes can be tricky. While it offers faster throughput and better hardware utilization, it often leads to training instability and worse generalization. To manage these risks, we explored several optimizers designed specifically for large-batch scenarios, aiming to find configurations that offer both stability and strong performance.

To help the models converge smoothly, we applied linear learning rate scaling, where the learning rate increases proportionally with the batch size. We also applied a warm-up strategy over the first 5-10 epochs to gradually ramp up the learning rate. This helps avoid overly aggressive updates at the start of training, which can otherwise cause instability when working with large batches. We also separated model parameters into two groups: those that should have weight decay (typically weights) and those that shouldn't (e.g., biases and normalization layers). This allowed us to apply LARS and LAMB more effectively by fine-tuning weight decay at a granular level.

4.1. SGDM

We evaluated Stochastic Gradient Descent with Momentum under large batch conditions using hyperparameters that had worked well in centralized training. SGDM showed a significant drop in performance as the batch size increased, highlighting its sensitivity to scaling.

4.2. LARS

LARS (Layer-wise Adaptive Rate Scaling) is specifically built to support large-batch training of deep networks. It dynamically scales the learning rate for each individual layer based on the ratio of weight and gradient norms. This mechanism helps maintain training stability by preventing updates that are too large or too small, which is especially important when working with deep architectures or large datasets.

In our setup, we incorporated the LARS optimizer from the torch-optimizer library using the same base configuration as SGDM and adjusted it to support larger batch sizes. This allowed us to observe its effect on both training efficiency and final accuracy.

4.3. LAMB

LAMB (Layer-wise Adaptive Moments optimizer for Batch training) is designed to make large batch training more stable and scalable by combining the adaptive gradient tracking of Adam with layer-wise learning rate normalization similar to LARS. This makes it particularly suitable for training deep neural networks on large datasets without degrading convergence or generalization.

In our implementation, we used the Lamb optimizer from the torch-optimizer library. The optimizer was used within the same training and evaluation loop as SGDM ensuring a consistent setup for comparison.

5. LocalSGD

In large-scale distributed training, frequent synchronization between workers can become a major bottleneck. To address this, LocalSGD introduces a strategy that reduces communication overhead by allowing multiple local updates before synchronization. Mini-batch Stochastic Gradient Descent (SGD) has long been the state-of-the-art in large scale distributed training due to its efficiency, however it struggles with network latency issues and bandwith limits. To overcome these challenges newer methods such as LocalSGD aims to lower the communication frequency by allowing each worker to perform multiple local updates before synchronization [5].

6. Experiments

For all experiments, we used a convolutional neural network based on LeNet-5 [4]. Our implementation enhances the original architecture by adding batch normalization layers after each convolutional block and dropout layers after the first and second fully connected layers to improve gen-

eralization. The full overview of the model can be found on our Github repository.

We evaluated the model using the CIFAR-100 dataset, which contains 60,000 color images of size 32x32 across 100 classes. We followed the standard data partitioning, using the predefined split of 50,000 training and 10,000 test images, without a separate validation set. This choice allowed us to maximize training data for our analysis. While a three-way split (train/validation/test) would provide more robust hyperparameter tuning, our primary objective was to compare the relative performance of different optimizers under similar conditions. The absence of a validation set does not compromise our results, as all methods were evaluated using the same train-test protocol.

We pre-processed the training set with common data augmentation techniques such as random cropping and horizontal flipping as well as normalization (both train and test sets) to minimize the generalization gap. The schedulers used in our experiments were the cosine annealing scheduler and the step scheduler to adjust learning rates dynamically allowing faster convergence and improved performance.

6.1. Centralized Learning: SGDM

To establish a strong baseline, we trained the model using SGDM (Stochastic Gradient Descent with Momentum), implemented via PyTorch's *torch.optim.SGD*. We included both momentum and weight decay in the optimizer configuration to improve convergence and generalization. A step-based learning rate scheduler was applied, with scheduled reductions at fixed milestones during training.

The training process was run for 100 epochs using a batch size of 256. As outlined in Section 6, we used the full training set for model fitting and evaluated performance on the predefined test split. A fixed random seed ensured reproducibility.

SGDM showed gradual but stable convergence throughout the training. The complete performance trend for SGDM, including accuracy and loss over time is illustrated in Fig. 1

6.2. Large Batch Optimization: LARS and LAMB

To explore how large-batch training affects convergence and performance, we evaluated two specialized optimizers: LARS and LAMB, both designed to maintain stability when using high-capacity models with large batch sizes. We increased batch sizes from 256 all the way up to 8192 and applied linear learning rate scaling based on a reference batch size, alongside warm-up and cosine annealing to encourage smooth convergence. In all cases, training was performed for 100 epochs.

Table 1 shows a comparison of the test accuracies of SGDM vs LARS vs LAMB for different batch sizes.

Batch Size	SGDM	LARS	LAMB
256	52.71%	52.91%	54.22%
512	49.06%	54.76%	54.93%
1024	44.61%	56.54%	54.54%
2048	38.76%	56.91%	53.68%
4096	29.70%	55.55%	50.98%
8192	19.74%	48.00%	46.00%

Table 1. Test accuracy (%) comparison of optimizers across various batch sizes.

6.2.1 LARS

LARS (Layer-wise Adaptive Rate Scaling) was implemented using the `LARS` class from the `torch-optimizer` library. We separated model parameters into two groups, those with weight decay (typically the weights) and those without (e.g., batch normalization and biases), to follow common best practices. The learning rate was scaled linearly with the batch size using a reference batch of 128, and a cosine annealing schedule was used following a short warm-up phase of 5 epochs.

LARS showed a significant improvement in convergence speed and accuracy over SGDM. Its layer-wise learning rate adjustment helped stabilize training despite the large batch size. Accuracy and loss improved steadily, and LARS consistently outperformed SGDM in terms of test accuracy under the same conditions.

6.2.2 LAMB

LAMB (Layer-wise Adaptive Moments for Batch training) extends Adam with a normalization strategy similar to LARS, making it highly effective in large-scale settings. We used the `Lamb` implementation from the `torch-optimizer` library. As with LARS, parameters were grouped to apply weight decay selectively.

LAMB used the same training pipeline and scheduler structure as LARS, including a warm-up phase for the first 10 epochs followed by a cosine annealing decay. However, the learning rate strategy required adjustment. During early experiments, we observed that using a high initial learning rate which was suitable for LARS led to unstable training or collapse when using LAMB. After hyperparameter tuning, we found that the best peak learning rate for LAMB ranged between 1×10^{-2} and 2×10^{-2}, depending on the batch size. Despite some small fluctuations in loss, LAMB maintained good generalization and stability throughout training.

6.3. LocalSGD

We performed data sharding to simulate the real environment of distributed training where each machine or

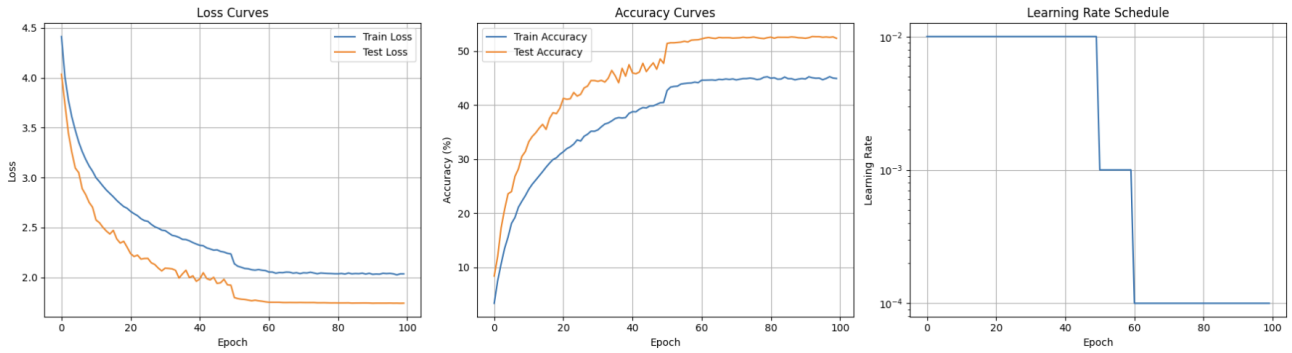

Figure 1. SGDM performance with batch size 256: Left – Training and Test Loss; Middle – Accuracy curves; Right – Step-based Learning Rate Schedule

node only has access to a portion of the dataset. This enforces a realistic communication constraint where workers only can compute gradients from their local data, and must then use the synchronization phase to share their parameters. Afterward we implemented LocalSGD [5] and evaluated its performance, experimenting with the numbers of workers K and local steps J. We used $K \in \{4, 12\}$ and $J \in \{10, 20, 30\}$

Table 2 presents the localSGD results.

J	K=4	K=12
10	29.75%	33.98%
20	35.66%	44.73%
30	39.75%	42.96%

Table 2. Performance comparison of LocalSGD across different values of local steps (J) and number of clients (K)

7. Discussion

Our experiments reveal important distinctions in optimizer behavior across different batch sizes. SGDM maintained stable convergence at smaller scales but failed to scale effectively, with accuracy dropping significantly as the batch size increased.

LARS showed the most consistent performance across the full range of batch sizes. While LAMB slightly outperformed LARS at smaller batch sizes (256 and 512), LARS became the clear winner from 1024 onward, achieving the highest test accuracy overall. One possible reason LAMB did not consistently outperform LARS may be due to differences in their design foundations. LAMB builds upon Adam's adaptive moment estimation, whereas our training pipeline was originally structured around SGDM. Future work could explore tuning LAMB in a setup more aligned with adaptive optimizers to fully leverage its capabilities.

LAMB still performed competitively but required more careful hyperparameter tuning. In particular, it was more sensitive to initial learning rate selection, with early instability observed when rates were too aggressive. Once tuned, however, it provided fast early convergence.

The LocalSGD results emphasized how distributed optimization benefits from reducing synchronization overhead. We observed that increasing the number of local steps (J) improves accuracy, especially when using more clients. This supports the notion that LocalSGD is a viable strategy for scaling distributed systems when bandwidth is limited.

8. Conclusion

In this work, we explored various strategies for optimizing deep neural network training under both centralized and distributed settings. Starting from a modified LeNet-5 architecture, we evaluated the performance of traditional and advanced optimizers including SGDM, LARS, and LAMB across a range of batch sizes. Our experiments showed that while SGDM offers stable convergence, its performance degrades significantly with larger batch sizes. In contrast, LARS and LAMB were more effective at scaling to large batches.

We also implemented LocalSGD to simulate a realistic distributed training scenario, evaluating its behavior under different configurations of worker count and local update steps. The results highlighted that increasing local steps can improve generalization while reducing synchronization overhead, especially with more clients.

Overall, our findings underline the importance of selecting the right optimizer and configuration when scaling training, particularly in distributed systems where communication costs and data partitioning introduce new challenges. Future work could investigate hybrid methods that combine communication-efficient techniques with adaptive optimizers, as well as applying these findings to larger and more complex architectures.

References

[1] Stitch et. Al. Don't use large mini-batches, use local sgd. In *ICLR*, 2020. 2

[2] You et. Al. Large batch training of convolutional networks. *arXiv*, 2017. 1

[3] You et. Al. Large batch optimization for deep learning: Training BERT in 76 minutes. In *ICLR*, 2020. 1

[4] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 2

[5] Sebastian U. Stich. Local sgd converges fast and communicates little. In *International Conference on Learning Representations*. ICLR, 2019. 2, 4