Distributed Learning of Deep Neural Networks

Sara Poiatti
Politecnico di Torino
s309616@studenti.polito.it

Luca Vigneri
Politecnico di Torino
s322925@studenti.polito.it

Anjali Narendra Vaghjiani
Politecnico di Torino
s328989@studenti.polito.it

Abstract

Distributed training is essential for scaling modern deep-learning models to larger datasets and architectures. This project focused on understanding and optimizing distributed training strategies using the LeNet-5 model on the CIFAR-100 dataset. We started by running centralized training to find the best hyperparameters for AdamW and SGDM optimizers, aiming to handle large batch sizes effectively. With these optimal settings, we tested how well AdamW, SGDM, and advanced optimizers such as LARS and LAMB performed with increasing batch sizes, identifying the largest batch size the model could handle without significantly decreasing the test accuracy. Afterward, we implemented LocalSGD and experimented with different numbers of workers and local iterations to find the best combination for distributed training. We extended our work by implementing the SlowMo framework to further analyze its potential to improve distributed training performance. Our findings provide practical insights into the optimization of distributed training setups, focusing on batch sizes, optimizer selection, and worker configurations for scalable and efficient machine learning. Finally, we developed an alternative to dynamically adjust the number of steps in the LocalSGD implementation. Our code is available on GitHub.

1. Introduction

Deep learning (DL) has achieved significant success in fields like computer vision, speech recognition, and natural language processing, driven by large-scale data and models with millions to billions of parameters. Despite advances in GPUs, training these models remains time-intensive. Data-parallel distributed training, particularly Stochastic Gradient Descent (SGD) and its variants, is widely used to accelerate training. Synchronous SGD with large batch sizes mitigates gradient variance but can degrade generalization performance, which led to incorporating techniques like square root learning rate scaling with warmup.

Advanced optimizers, including Layer-wise Adaptive Rate Scaling (LARS), AdamW, and LAMB, were tested on the same dataset to address the limitations of SGD with large batches. LARS, which adapts learning rates per layer, significantly improved training stability and efficiency. Despite these advancements, large batch training often reduces generalization performance.

To address this, methods like LocalSGD and the Slow Momentum (SlowMo) framework were explored. LocalSGD balances computational and communication costs by allowing workers to perform multiple local updates before synchronization. SlowMo extends this by incorporating momentum-based updates for improved convergence. A dynamic adjustment mechanism was introduced to reduce synchronization frequency, addressing communication overhead and enhancing efficiency. These techniques provide insights into optimizing large-scale distributed training.

For all experiments, we used a LeNet-5 model [6] and the CIFAR-100 dataset. As the original dataset lacked a dedicated validation set, we split it into training and validation subsets in an 80/20 ratio. The validation set was specifically used for hyperparameters tuning. The training set was preprocessed with data augmentation techniques [1], such as random cropping and horizontal flipping, to enhance generalization. Additionally, we used the cosine annealing scheduler to adjust the learning rate dynamically during training, facilitating faster convergence and improved performance.

2. Related Work

Distributed optimization is vital for large-scale machine learning but faces challenges like communication overhead and synchronization delays. Training with extremely large batches is difficult, often requiring careful hyperparameters tuning to maintain performance.

Optimizers like LARS and LAMB [2] [3] address these challenges. LARS adapts learning rates layerwise, while LAMB combines per-dimension and layerwise normalization for better convergence in nonconvex problems. LocalSGD [4], compared to mini-batch SGD, achieves lower training loss and better test accuracy, with post-local

SGD further improving generalization by transitioning from large-batch to local updates later in training.

Slow Momentum (SlowMo) [5] reduces communication overhead in distributed SGD by allowing multiple local updates before synchronization and applying momentum updates during synchronization. This approach enhances communication efficiency and convergence stability, proving effective for tasks like image classification and machine translation. These advancements inform strategies for optimizing large-scale distributed training.

3. Centralized Learning

In centralized learning, the choice of the optimization algorithm significantly impacts the model performance, convergence speed, and generalization ability. This section explains the general behavior of Stochastic Gradient Descent with Momentum (SGDM) and AdamW in the context of centralized learning, both of which were used in this study using the PyTorch framework.

3.1. Stochastic Gradient Descent with Momentum (SGDM)

SGDM extends standard Stochastic Gradient Descent (SGD) by introducing a momentum term that accelerates convergence and reduces oscillations during training. Momentum stores a moving average of past gradients, enabling the optimization to make larger steps in directions where the gradients are consistently pointing. This helps mitigate oscillations, especially in high-dimensional parameter spaces, leading to faster convergence and improved stability. SGDM is implemented in PyTorch using torch.optim.SGD optimizer, where the momentum coefficient is set as a hyperparameter.

3.2. AdamW: Decoupled Weight Decay

AdamW is a variant of the Adam optimizer that decouples weight decay from the gradient update process. This separation ensures that regularization is applied independently of optimization, leading to better generalization and more stable learning, especially in deep neural networks.

In PyTorch, AdamW is implemented using the torch.optim.AdamW optimizer, where both the learning rate and weight decay are adjustable hyperparameters.

4. Large batch optimizer

Training deep neural networks with large batch sizes presents unique challenges, such as potential instability and degraded generalization. To address these challenges, we explored various optimizers and strategies tailored for large-batch optimization. The goal was to evaluate the performance of different optimizers and identify optimal configurations for stable and efficient training.

The learning rate was adjusted based on the square-root scaling rule to maintain stability and convergence, as proposed in prior works [3]. Additionally, we used a warm-up phase of 5 epochs to gradually increase the learning rate, helping to avoid large updates that could destabilize the training process in the early stages. This approach ensured that the optimizer could handle the increased batch size without compromising training dynamics.

4.1. SGDM and AdamW

We implemented SGDM and AdamW for large-batch optimization, starting from the best parameters identified through centralized training. The results demonstrated a clear challenge with large batch sizes when using SGDM, as its performance dropped significantly as the batch size increased.

4.2. LARS

LARS (Layer-wise Adaptive Rate Scaling) is an optimizer designed for training large models with big batch sizes. It works by adjusting the learning rate for each layer individually, based on the layer's weight and gradient norms. This helps prevent issues like vanishing or exploding gradients, making training more stable and efficient, especially for deep networks and large datasets. The implementation is shown in Algorithm 1 in paper [3].

In our experiments, we implemented LARS for large-batch training, starting from the best hyperparameters of SGDM found during centralized training.

4.3. LAMB

LAMB (Layer-wise Adaptive Moments for Batch training) is an optimization algorithm designed for training large deep learning models using large batch sizes. LAMB adapts the learning rate on both the gradient information and a layer-wise scaling strategy, similar to LARS, addressing varying magnitudes of gradients across different layers. Combining elements from Adam and LARS, LAMB ensures stable and efficient training, improving convergence and generalization. This method adjusts the learning rate and moment estimates for each layer, resulting in faster convergence and reduced gradient update variance, as detailed in [3], Algorithm 2.

We implemented LAMB for large-batch training, starting from the best hyperparameters of AdamW found during centralized training.

5. LocalSGD and SlowMo

Distributed training techniques aim to reduce communication overhead and improve the scalability of training deep learning models.

5.1. LocalSGD

One such technique is Local Stochastic Gradient Descent (LocalSGD) [4], where each worker performs multiple local updates before synchronizing with others. This method reduces the frequency of communication, making it especially suitable for large-scale distributed systems. We implemented and evaluated LocalSGD to determine its effectiveness across varying configurations of local iterations and worker counts.

We experimented with different combinations of K (number of workers) and J (number of local steps) with $K \in \{2, 4, 8\}$ and $J \in \{4, 8, 16, 32, 64\}$, to analyze the trade-offs between communication efficiency and model performance. The learning rate was scaled by the number of workers (K). Results can be seen in Table 3

5.2. SlowMo

Another approach relies on the use of two optimizers: one operating in an outer loop and the other in an inner loop, inspired by the Slow Momentum (SlowMo) framework proposed by Wang et al. [5]. SlowMo was designed to improve communication efficiency in distributed settings by introducing less frequent global momentum updates, mitigating bottlenecks caused by synchronization among workers.

The SlowMo framework extends basic optimization algorithms, such as Local Stochastic Gradient Descent (LocalSGD), by introducing a two-phase approach. In the first phase, local updates are performed: each worker executes multiple iterations on a local shard of the dataset. In the second phase, global synchronization and momentum updates take place: after a predefined number of local iterations, the workers synchronize through a global communication operation and update the shared momentum.

This approach was conceived to address the communication efficiency problem typical of the traditional distributed paradigm, where workers compute gradients locally on mini-batches and aggregate them using an ALLREDUCE operation before each optimization step. The mandatory completion of ALLREDUCE at every iteration represents a significant bottleneck, as synchronization is sensitive to stragglers.

The framework operates according to Algorithm 1 from paper [5], introducing α and β, two hyperparameters that control the impact of the momentum buffer and the momentum decay rate, respectively.

Unlike the original SlowMo framework, where each worker maintains a local copy of the momentum buffer, in our implementation, we use a single global buffer. This is possible because, since we are unable to work in parallel, it is not necessary for each worker to have an independent copy of the buffer.

6. Dynamic Local steps adjustments

In distributed training, the number of local steps J (between synchronization events) significantly impacts both communication efficiency and model performance. A fixed J, while simple to implement, can lead to suboptimal trade-offs:
- **Increasing J**: Reducing communication frequency decreases overhead but introduces bias into the gradient estimate. This happens because local updates drift further apart without synchronization, potentially leading to divergence.
- **Reducing J:** Synchronizing more frequently improves convergence by ensuring consistent parameter updates but increases communication overhead, limiting scalability.

By dynamically adapting J based on the gradient magnitude and variance, we aim to strike a balance between convergence speed and communication efficiency. This approach ensures that the system adjusts to the changing dynamics of the training process and accounts for heterogeneity across workers.

To effectively adapt J, two key gradient statistics are monitored during training: i) Gradient Magnitude G(t) represents the strength of gradients at time t. A large G(t) implies that model parameters are undergoing significant updates, suggesting that workers can afford to perform more local steps J without risking divergence. ii) Gradient Variance $\sigma^2(t)$ measures the variability of gradients across workers at time t. High variance indicates that workers are observing different gradients, which can cause divergence if local updates are prolonged. In such cases, smaller J values are needed to synchronize more frequently and maintain consistency.

We propose a heuristic rule for dynamically adapting J based on G(t) and $\sigma^2(t)$:

$$J_t = min(J_{\max}, max\left(J_{\min}, \alpha \cdot \frac{G_t}{\sigma_t^2 + \epsilon}\right))$$

Where:
- $J_{\min}$ and $J_{\max}$ are the minimum and maximum allowable values for J, ensuring bounded behavior
- α is a scaling factor that controls the sensitivity of J to the gradient statistics
- ϵ is a small constant to prevent division by zero.

When G(t) is large and $\sigma^2(t)$ is small J_t increases, allowing workers to perform more local steps, reducing communication frequency and improving efficiency. When G(t) is small or $\sigma^2(t)$ is large J_t decreases, leading to more frequent synchronization. This is necessary to mitigate divergence caused by inconsistent gradients or weak updates.

The proposed dynamic J adjustment can be seamlessly integrated into the LocalSGD algorithm as follows:
i) Gradient Statistics Calculation [7]: At each synchro-

nization step t, calculate the gradient magnitude G(t) and variance $\sigma^2(t)$ using:

$$Gt = \|g_t\|^2 - \frac{1}{k} \cdot \sigma_t,$$

$$\sigma_t^2 = \frac{1}{k-1} \sum_{i=1}^{K} (\|g_t^k - g_t\|^2,$$

Where:
- $g(t)^k$ is the gradient computed by worker k at time t,
- g(t) is the average gradient across all workers.

ii) Update J_t: Use the heuristic rule to calculate the number of local steps J_t for the next interval.

iii) Perform J_t Local Steps: Workers perform J_t local updates independently on their respective data shards before synchronizing.

iv) Synchronize: After J_t local steps, all workers synchronize their parameters using the ALLREDUCE operation and repeat the process.

7. Experiment

For all experiments, we have used the LeNet-5 model [6] on CIFAR-100 dataset with the cosine annealing scheduler. Training was carried out for 150 epochs.

7.1. Centralized Learning: SGDM and AdamW

This section aims to evaluate the performance of the optimization algorithms SGDM and AdamW in terms of their ability to achieve high accuracy and low loss during the testing phase. For hyperparameters tuning, it is important to note that the CIFAR-100 dataset does not include a dedicated validation set. Therefore, we split the original training set into 80% for training and 20% for validation to perform hyperparameters tuning, setting a seed for the reproducibility of the random operation. After identifying the best hyperparameters, we re-trained the models using the full training set and evaluated their performance on the test set to achieve the best possible accuracy and minimum loss. The results we obtained are shown in table 1 and the accuracy and loss plots are shown in Figure 1 and Figure 2.

The results show that AdamW converges faster than SGDM, with a rapid reduction in loss and an increase in accuracy during the early epochs. However, AdamW exhibits more noticeable oscillations in both loss and accuracy, indicating less stability. In contrast, SGDM converges more slowly but with greater stability, displaying smoother and more predictable curves. In summary, AdamW is better for fast convergence, while SGDM ensures greater long-term stability.

7.2. Large Batch Optimizer: LARS, LAMB, SGDM, Adamw

Four optimizers were tested: SGDM, AdamW, LARS, and LAMB. Each optimizer's learning rate was scaled based on the square-root scaling rule [3] relative to a reference batch size of 64. To adjust the learning rate during training, we used a custom scheduler that included a linear warm-up phase for the first five epochs, followed by a gradual reduction over the remaining epochs.

For SGDM and AdamW we have used the same hyperparameters shown in Table 1. For LARS, we tuned the hyperparameters starting from those of SGDM, obtaining a learning rate of 25e-2 and a weight decay of 4e-4. For LAMB, we tuned the hyperparameters starting from those of AdamW, obtaining a learning rate of 5e-4 and a weight decay of 1e-1.

The performance of each optimizer was evaluated across various batch sizes using test accuracy. Key observation for this were shown in Table 2:

SGDM:- It struggled to go beyond batch 512, where generalization degraded significantly.

AdamW:- When the same test was executed using AdamW we show that it offered better performance than SGDM and could reach efficiently till batch size 8K.

LARS:- Achieved great results even for the large batch sizes as 8K.

LAMB:- This showed the best results from all the optimizers with the best accuracy for very large dataset as batch size 8K.

7.3. LocalSGD + SlowMo

We extended our experiment to the distributed scenario by implementing LocalSGD and SlowMo, simulating the parallel work of K workers with a serialized implementation. In both cases, each worker receives a local shard of the dataset of size

$$\frac{\text{length of dataset}}{K}$$

which remains identical in every test due to a fixed seed ensuring the same random partitions each time. To perform as many iterations as in the centralized scenario when increasing the number of local steps (J) or the number of workers (K), the total number of iterations in our framework was made equivalent:

$$K \cdot J \cdot \text{global\_iteration} = \text{centralized\_iterations}$$

Where K is the number of workers, J is the number of steps in the inner loop, global_iteration is the number of iterations in the outer loop and centralized_iterations is the

Optimizer type	Learning rate	Weight decay	Test accuracy [%]	Test loss
SGDM	1e−3	4e−4	51.47%	1.9361
AdamW	1e−4	1e−1	52.86%	1.8173

Table 1. Show the test accuracy [%] and test loss with the best hyperparameters. For all experiments, we have used epochs=150, batch size=64.

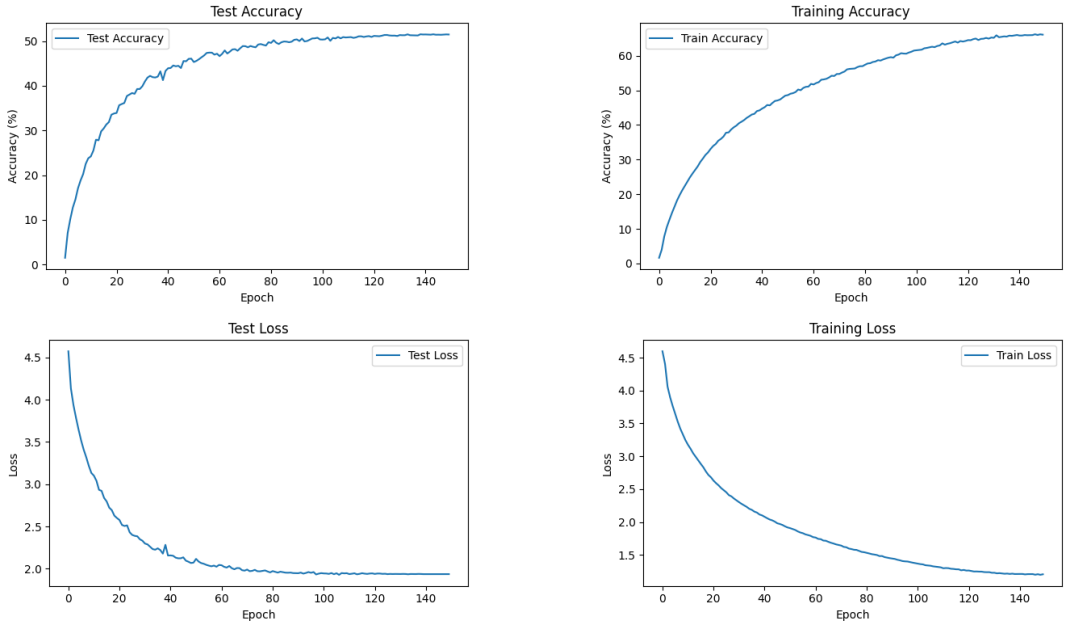

Figure 1. SGDM: Compare the convergence behaviour of the training set and test set.

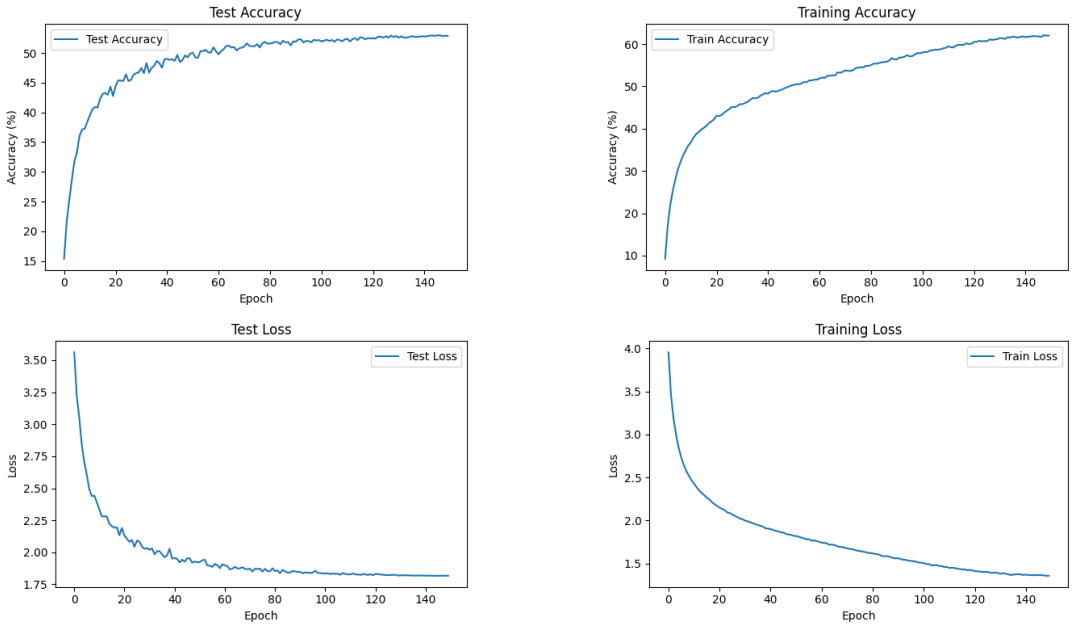

Figure 2. AdamW: Compare the convergence behaviour of the training set and test set.

Batch Size	128	256	512	1024	2048	4096	8192
AdamW	50.40%	49.05%	48.82%	47.99%	46.69%	45.94%	43.60%
SGDM	49.48%	43.44%	38.75%	-	-	-	-
LARS	48.16%	46.98%	46.01%	44.54%	42.81%	39.33%	35.77%
LAMB	48.79%	47.55%	47.72%	48.59%	47.23%	46.03%	43.70%

Table 2. Shows the test accuracy % for all optimizers for different batch sizes.

number of total iterations in the centralized scenario. In our case, the number of iterations in the centralized scenario per epoch is equal to:

$$\frac{\text{len(dataset)}}{\text{batch size}} = \frac{50,000}{64} = 781.25 \approx 782$$

We compute the computation and communication time for all the combinations of workers and local steps to analyze the differences between them.

All experiments in this section have been done using the same hyperparameters as SGDM shown in Table 1.

7.3.1 LocalSGD results

As J increases, workers perform more local updates before synchronizing, which reduces communication overhead significantly. For instance, in the data provided, communication time drops from 11.37 s to 0.88 s when J increases from 4 to 64 for K = 2. This is beneficial in systems where communication is a bottleneck, such as those with high latency or limited bandwidth.

However, the downside is that computation time increases because workers are doing more work locally. For example, with K = 8, computation time increases from 3521.66 s to 4953.34 s as J growth. This can also delay synchronization, potentially slowing convergence or leading to inconsistencies if local updates diverge too much. The differences in communication and computation time considering the minimum and maximum values of J are shown in Table 4.

In summary, increasing J reduces communication time but increases computation time. The choice of J depends on the system's characteristics: higher J is better for communication-constrained setups, while lower J suits systems with high computational capacity and efficient communication.

The results show that the best combination in terms of test accuracy is K = 2 and J = 4, but considering communication time it is the worst (11.37 s).

7.3.2 SlowMo results

The hyperparameters α, β, and J play a crucial role in the framework's effectiveness and must be carefully chosen. To tune these hyperparameters, we use the best hyperparameters identified for SGDM in the centralized scenario. We found that the best values for our model are $\alpha = 0.75$, $\beta = 0.20$, and $J = 16$. For values of $J > 16$, we observe that the model tends to overfit.

For values of $\alpha = 1$, $\beta = 0$, $K = 1$, and $J = 1$, we obtained results that, as expected, are almost identical to those of centralized SGDM with epoch = 150 and batch size = 64.

The behavior of SlowMo using different values of J and K is similar to LocalSGD. Table 5 shows the results of various combinations.

7.4. Dynamic local steps adjustment for LocalSGD

We implemented dynamic J adjustment in LocalSGD and evaluated its performance on the CIFAR-100 dataset with the LeNet-5 model. The results demonstrated improved convergence rates and reduced communication overhead compared to fixed J configurations, validating the effectiveness of the proposed approach.

For all experiments, we have used the best hyperparameters identified for SGDM in the centralized scenario Table 1.

As α increases, the computation time slightly increases, leading to more local computation per communication round. This is consistent with the intuition behind dynamic local step adjustments: higher α reduces communication frequency but increases the local computation burden.

Communication time significantly decreases as α increases. Larger α reduces the number of communication rounds because the model is updated locally more often before synchronizing. This is a key benefit of LocalSGD for distributed learning when communication is a bottleneck.

Test accuracy and test loss are consistent across all α values, meaning that the dynamic local step adjustment mechanism is robust across a range of α values. The main benefit of increasing α is the significant reduction in communication time, while the primary cost is a slight increase in computation time. This makes the method suitable for distributed training environments where communication is a bottleneck.

8. Conclusion

Through our work we can assess that in the centralized scenario SGDM is more stable than AdamW but AdamW reaches higher values in the first epochs, while at the end of

J	K=2	K=4	K=8
4	52.28%	51.37%	51.31%
8	51.64%	51.54%	50.10%
16	51.86%	51.46%	50.72%
32	52.15%	51.63%	51.29%
64	51.92%	51.41%	51.49%

Table 3. Test accuracy results from the combination of K and J in a distributed environment.

K, J Combination	Computation Time [s]	Communication Time [s]
K=2 J=4	3482.30	11.37
K=2 J=64	4148.00	0.88
K=4 J=4	3482.01	8.97
K=4 J=64	4436.54	0.74
K=8 J=4	3521.66	7.70
K=8 J=64	4953.34	0.68

Table 4. Evaluation of computation and communication cost in a distributed environment.

K, J Combination	Computation Time [s]	Communication Time [s]	Test Accuracy [%]	Test Loss
K=1 J=1, $\alpha = 1.00$, $\beta = 0.0$	3611.01	154.65	51.79%	1.9487
K=1 J=16, $\alpha = 1.00$, $\beta = 0.0$	3780.73	10.26	51.76%	1.9452
K=1 J=16, $\alpha = 0.75$, $\beta = 0.2$	3720.44	9.97	51.43%	1.9359
K=2 J=16, $\alpha = 0.75$, $\beta = 0.2$	3757.84	6.00	51.65%	1.9283
K=4 J=8, $\alpha = 0.75$, $\beta = 0.2$	3496.41	7.22	50.92%	1.9397
K=4 J=16, $\alpha = 0.75$, $\beta = 0.2$	3765.07	3.99	51.45%	1.9470
K=8 J=16, $\alpha = 0.75$, $\beta = 0.2$	3944.41	3.07	50.53%	1.9653

Table 5. Results from the combination of K and J in SlowMo framework.

α Value	Computation Time [s]	Communication Time [s]	Test Accuracy [%]	Test Loss
100	3572.42	7.64	51.02%	1.9665
500	3577.99	3.74	51.26%	1.9584
700	3725.85	3.33	51.19%	1.9490
1000	3764.15	2.70	51.76%	1.9258
1500	3844.46	2.25	51.07%	1.9577
3000	3910.57	1.55	51.07%	1.9430
4500	3850.40	1.43	51.05%	1.9354

Table 6. Results for $K = 4$, $\epsilon = 1 \times 10^{-6}$, $H_{\text{min}} = 4$ and $K_{\text{max}} = 64$ with different α values to analyze the behavior of the dynamic steps adjustment.

the training they reach almost the same values.

Our large-batch training experiments showed that SGDM struggled with generalization beyond a batch size of 1024, while AdamW maintained better performance. Their large batch counterparts, LARS and LAMB, perform better. Lamb got the best results out of the four optimizers.

In the distributed scenario, from the analysis of the results presented in Table 3 and Table 5, it can be observed that the values of accuracy and loss in the different configurations are almost identical. However, what actually varies is the computation and communication time. When analyzing LocalSGD and SlowMo, it is observed that the latter, when comparing the plots with $K = 4$ and $J = 16$, introduces less noise due to the presence of the double optimizer. However, this advantage comes at the cost of increased computation and communication times.

Regarding our personal contribution to the project, we decided to adjust the number of local steps dynamically in LocalSGD. We have not achieved a significant improvement in the model accuracy and loss but this new mechanism helped the model to slightly reduce noise during the training and testing phases.

References

[1] Reddi et al. *Adaptive Federated Optimization*. In *ICLR 2021*. 1

[2] You et al. *Large Batch Training of Convolutional Networks*. In *arXiv 2017*. 1

[3] You et al. *Large Batch Optimization for Deep Learning: Training BERT in 76 Minutes*. In *ICLR 2020*. 1, 2, 4

[4] Stich et al. *Don't Use Large Mini-Batches, Use Local SGD*. In *ICLR 2020*. 1, 3

[5] Wang et al. *SlowMo: Improving Communication-Efficient Distributed SGD with Slow Momentum*. In *ICLR 2020*. 2, 3

[6] Hsu et al. *Federated Visual Classification with Real-World Data Distribution*. In *ECCV 2020*. 1, 4

[7] Balles et al. *On the Choice of Learning Rate for Local SGD*. In *TMLR 2024*. 3