Exploring Centralized and Distributed Learning with Optimizers : A study using CIFAR-100 for AML Project

Yeganeh Nouri
S329541

Giulio Titonel
S333891

Mostafa Kordi
S329702

Abstract

Advanced machine learning techniques often require great computational resources. A solution to this problem is the use of distributed learning techniques. We used modified LeNet-5 architecture and conducted experiments with tuning the hyperparameters for evaluating the impact of learning rates, weight decay and batch sizes on model performance. We have done experiments with two different types of scaling criteria: large batches and distributed training. We compared the performance of different optimizers such as SGD, AdamW, LARS and LAMB into large batches and distributed learning setups. We used the CIFAR-100 dataset and modified LeNet-5 architecture for all the optimizers and all the experiments. To do so, we compare the results on a large batch and distribute setup with the centralized scenario. Large batches were tested with SGD, AdamW, LARS and LAMB. Distributed learning was conducted using LocalSGD with periodic weight averaging, then we improved using SlowMMo for better convergence. We have found that we can scale up to batch-size=256 in SGD, 1024 in AdamW, 512 with LARS and 2048 with LAMB. We have found that there are no significant changes scaling the number of local workers and the number of local steps before averaging in localSGD and slowMMO nor improved and neither change the results. Finally, we have tried to see what happens if the dataset is splitted randomly to experiment a real-case scenario: we simulated a situation where the local workers have different computational power and don't wait for each other until the end of the J steps. Instead, they send the value every fixed time, without waiting for the others to finish the training. To simulate this in our code, we gave different J steps to each local worker. We have found that, in this scenario, some problems occurred and so we added a average weighted on the J local steps to fix the problem. We also tried to use slowMMO. However, the results didn't improve.

1. Introduction

Neural networks are essential for modern AI, but training them on large datasets is a challenge. This project simulates distributed learning to handle complexity of tasks and size of datasets. We are simulating the parallel training by dividing the training process across multiple machines or devices, based on the algorithm from Stich et al. [1] . Each device handles a portion of the data, and their work leads to training the model. With distributed learning we can train the model faster and enable efficient usage of computational resources that we have. This factor is important for real-world applications that require high computational power and training time, such as NLP and computer vision. In this project, our focus and goals were on:

- Find in the centralized scenario the best learning-rate and weight-decay
- Comparing centralized vs. distributed methods
- Exploring optimizers like LARS and LAMB
- Testing on large-batch optimizers and tuning the hyperparameters such as learning rate and weight decay, based on the batch-sizes
- Analyze how much we can reduce computation and communication costs without convergence issues
- Analyze a scenario where the computational resources are distributed unequally on the different workers. Try to reduce the synchronization waiting time in case of different available computational power on the workers. Try to improve results in case of problems.

In the Baseline Centralized Training setup, we train the model on a single node using SGD and AdamW optimizers on the CIFAR-100 dataset. vs. in a distributed setup we trained the model across multiple nodes and implemented it with LocalSGD and SlowMo.

Our project is built on methodologies mentioned in the 'Previous Work' section, by implementing and testing optimizers like SGD, AdamW, LARS, and LAMB; we provided a comparison of their performance with different hyperparameter configurations.

2. Previous and Related Work

2.1. Large Batch Training

One way to improve resource exploitation is by using larger batches. Many attempts have been done in the past for scaling optimizers for large datasets without accuracy loss. You et al. [3] came up with LARS, proposing layerwise learning rate adjustments, and then came up with LAMB [4], adding adaptive moment estimation. The improvements were impressive. However, we want to test them on a LeNet-5 network. All those methods used a warm up to stabilize the gradients first.

2.2. Local SGD

Stich et al. [1] proposed that LocalSGD can be an alternative to standard SGD for distributed learning. Models will be trained locally for several steps before averaging their weights. In this way it will be effective in reducing communication costs. LocalSGD requires tuning of the number of local steps (J) and the number of workers (K) to avoid loss of generalization. Yu et al. [5] proposed momentum-based variants of SGD for distributed learning, with analyzing the trade-offs between computation and communication efficiency. They believe by combining momentum and local updates, this method will achieve near-linear speedup, with significant tuning to balance momentum between workers.

2.3. Optimizers for Distributed Learning

Wang et al. [2] presented SlowMMO, an optimization technique that combines localSGDM with momentum-based updates of the weights. In this approach, there is an internal momentum related to SGD and an external momentum related to the average. This last one is applied instead of the simple average of the gradients. Authors examined momentum in distributed training, showing its effectiveness in reducing oscillations and improving stability during optimization. While it is effective for certain applications, it is also complex in implementation and parameter tuning, so it will limit its adoption in distributed systems.

3. Method

Our defined method was learning about theoretical concepts in distributed learning then adding practical implementation and executing distributed training experiments. This methodology enabled experimentation by providing a framework for it, aligned the theoretical concepts with practical implementations. So we can optimize and explore the performance of centralized and distributed learning and analyze their trade-offs in training.

These are the approach we took and structured into the following steps:

3.1. Development of Codebase

The codebase is written in Python. The logic of the model was implemented using PyTorch with torchvision for dataset handling. We also used LeNet5 model architecture for CIFAR-100 dataset handling and transformation such as flipping. Our codebase is designed to support centralized and simulated distributed learning setups. Training, validation, and evaluation set were selected from the CIFAR-100 dataset. For defining the loss function we used `torch.nn`. For optimization techniques, we used `torch.optim` and `torch.distributed` for distributed training. All the hyperparameters (lr, weight-decay, momentum, optimizer, etc.) are passed through the command line, so that every experiment is always run on the same code.

Launching the code and passing the parameters from command line, the following things happen in broad term: The CIFAR-100 dataset is divided into 3 partitions: training, validation and test set. All those data are first transformed applying dimension-wise normalization. The division has a seed, so that the dataset is always divided in the same way. This is useful when we have to re-divide the dataset because of a resources' shortage without starting from scratch. A csv is created. This is useful to save the advancement and to build a graph. Then the model starts to learn from the dataset. For each epoch it runs K (given by command line) different local models for J local steps. After learning for all local models, the local models are averaged and then the train restarts until the end of the dataset and then restart with another epoch, rescales the lr, ecc. until the given number. There are some extra parameters that sets a warmup (so the lr will be first increased following warm up rules), allows to test the "unequal computation power" case, for slowMMO, ecc. Every 5 epochs, the last model and the last csv is saved. There is a function to build a graph of losses and accuracies (both training and validation) starting from the generated csv.

And there are the following technical parameters used:

- K=number of local workers; J= numbers of local steps. If K=J=1 we are in centralized situation epochs=number of total epochs. After experiments we set to 200
- batch-size: 64 for standard, increased for large batch. Increase the batch-size automatically increase the lr!
- Momentum: default at 0.9. We did very few tries to change it but they were not successful neither useful.
- Lr-max and lr-min: the max and min lr of cosine annealing

- Weight-decay
- Optimizer: can be sgd, adam, lars or lamb
- warmup: number of warmups. 0 if none.
- Trust-coefficient and beta-2: data useful for lamb

3.2. Centralized Baseline

In the first step of exploring distributed learning approaches and before testing distributed methods, we wanted to provide a comparison for distributed learning. Hence, we implement centralized training baseline using the LeNet5 model architecture which is trained on the CIFAR-100 dataset using SGD (Stochastic Gradient Descent) and AdamW as the default optimizer, in training we used 'learning rate, batch size, epochs' as our defined hyperparameters. The lr was scheduled using cosine annealing, with final value=0. These hyperparameters were tuned to maximize baseline performance and ensure a meaningful comparison later. With our setup we were able to experiment and compare distributed methods in terms of accuracy, loss and training. Note that in centralized training, we trained the model on a single machine.

3.3. Large Batch

Then we moved with SGD and AdamW on large batches. We gradually increased the batch size until we found that the model started to worsening its performances significantly. We applied a warm-up on the lr for 5 epochs, during which the lr rises progressively. We scheduled the learning rate after warm up using the squared ratio between 64 and the new batch size, as proposed by You et al. [4]. Then, we tried with LARS and LAMB to increase the batch size further without losing performances. The expectation is that we can scale SGD and AdamW until a certain batch size, and then LARS and LAMB to a higher batch size without losing accuracy.

3.4. Simulated Distributed Learning

For simulating distributed learning, we divided the dataset across multiple 'virtual workers' and trained them sequentially using the LocalSGD approach. To simulate the real-world scenarios in parallel training, we had to split the data. It means data is divided into partitions across multiple nodes; the CIFAR-100 dataset was divided among K (number of workers). For simulating the communication step and reducing the communication cost, weight averaging was performed every J (number of local steps). Adding K and J parameters allowed experimentation with trade-offs between computational efficiency and communication overload. IID sharding splits the dataset into K parts (e.g., $K = \{2, 4, 8\}$). LocalSGD performs J local updates before synchronizing (e.g., $J = \{2, 4, 8, 16, 32\}$). Then we used SlowMMo to improve the performances. On a mathematical point of view, we should expect similar results to large batch size: the ratio between the maximum batch size reached and 64 should be equal to the product of K and J. However, large batch scaling uses a higher lr and this can easily lead to instability, even if we use the warmup. So we expect that LocalSGD gives better results than local batch scaling.

3.5. Local workers with different computational power

A realistic scenario in distributed learning is when the workers have different computational power. In this situation, waiting for all the workers to end their local steps can be very expensive, as the slowest worker is a bottleneck. To do so, we could expect the workers to synchronize after a fixed time rather than after J local steps. This implies that every worker does a different number of local steps before averaging. This should worsen the results, for two main reasons: first, some workers do a very high number of local steps, and secondly all the workers have the same relevance in the computation of the mean despite some having done less steps. To address a possible solution for this second problem, for the averaging phase, we introduced an average weighted for the number of local steps done. We expect that this method resolves the problem. Then we wanted to try to see if slowMMO can improve degradation or not. We expected it to improve degradation because it reduces the impact of the bad local updates.

3.6. Training and Optimization

We implement a training part addressing computational constraints and project objectives. For conducting experiments and training the model, we used optimizers such as: SGD, AdamW, LARS, and LAMB. They represent optimization techniques for deep learning. SGD (Standard optimizer for convergence), AdamW (Weight-decay optimized Adam variant), LARS, and LAMB (Designed for large-batch training and distributed setups).

The hyperparameters that we tuned and changed in every experiment, were: Learning rates, batch sizes, and weight decay, J and K. First we found the optimal lr and weight decay in the centralized scenario: Then we adjusted batch size to test on large batches. Finally we adjusted K and J to apply SGD. The learning rate (lr-max, 0) was scheduled using cosine annealing to ensure stable training. Batch sizes were varied (64, 128, 256, 512 and 1024) to study their impact on generalization and stability. Loss function and test accuracy was the output of our experiments while we were training the model.

3.7. Training Progression and Metrics

We kept track and logged the progression during the training process to monitor trends and adjust parameters if necessary, by tracking metrics and logging both accuracy and loss across training and validation datasets, which helped us analyze the performance later on. And by doing that we noticed periodic evaluation that ensured insights into our model's performance. Periodic evaluation of test and training accuracy helped us to detect overfitting or underfitting and identify when to stop training early.

3.8. Evaluation Metrics

We evaluate the performance of centralized and distributed training by using these metrics:

- Test Accuracy (measured generalization performance at the end of the training)
- Train accuracy (tracks learning progression)
- Train loss (monitor progressions in the model for each epoch)
- Validation Loss (used for monitoring convergence and detect overfitting, a decreasing validation loss indicates convergence)
- Validation accuracy (tracks accuracy on the validation set)
- Communication overhead (the frequency of weight averaging (J) was analyzed for balancing between computational efficiency and communication cost)
- Scalability (we explored the impact of increasing the number of workers (K) and batch size on model performance)
- Saving the last model (automatically saves the last model every 5 epochs ensuring we don't lose it during long training sessions)

3.9. Visualization

For measuring the model performance visually after 5 epochs the model will create CSV file capturing metrics used for analysis. Training and validation loss/accuracy curves were generated to analyze performance.

4. Experiments and Analysis

The goal of the experiments was training ML models with 'centralized and distributed learning' setups and comparing their performance. We also used different experimental setups or methodological aspects which were interconnected to the project objectives, to evaluate distributed learning techniques and compare them to centralized training. Each setup explores unique challenges and solutions related to distributed learning.

Table 1. Centralized Training Results

Optimizer	Parameters	Accuracy
SGD	lr=0.04, wd=0.005	51.85%
AdamW	lr=0.0002, wd=0.04	49.74%

Table 2. Centralized SGD Results

lr-max	lr-min	Weight Decay	Momentum	Validation Accuracy
0.01	0.001	-	0.9	48.02%
0.005	0.0005	0.0001	0.9	42.76%
0.002	0.0002	-	0.9	33.94%
0.02	0.002	0.0001	0.9	50.48%
0.01	0.001	0.00001	0.9	48.77%
0.03	0.003	0.0005	0.9	50.39%
0.03	0.003	0.001	0.9	51.53%
0.03	0.003	0.003	0.9	49.99%
0.04	0.001	0.005	0.9	51.26%
0.001	0.001	0.005	0.9	50.43%
0.04	0.0	0.005	0.9	51.85%

4.1. Experiment 1: Centralized Training

This was the baseline setup in which all training was performed on a single node, and helped us to compare distributed setups. Experiments on this setup established how the model performs in simple conditions, and we used the centralized training baseline to find the optimal lr and weight-decay for SGD and AdamW. We used cosine annealing to schedule a progressive reduction of the learning rate. Figures of the accuracies and losses obtained during the training for the best hyperparameters are available at figures 1, 2, 3, 4. Furthermore, some of the parameters we tried are available with the respective validation accuracy in table 2 and 3 (the test accuracy is used only after deciding the hyper-parameters).

To do so, we run the same experiment with these parameters below, in order to find the ones that get the best final test set accuracy. See Table 1

4.2. Experiment 2: Large Batch Optimizers

We used the same lr and weight-decay found for SGD and AdamW. We only increased the batch-size and scaled the learning rate based on the squared ratio between SGD and AdamW, but adding the first 5 epochs of warmup, following [4]. We have pushed the increase of batch size until we started seeing degradation. Then we went on with optimizers which are specialized for scenarios where very large batch sizes are used in training, such as LARS and LAMB, they are part of the distributed setups. For SGD the maximum working size was 256, for AdamW 512, for LARS 512 and for LAMB 1024. Things didn't go as we expected,

Table 3. Centralized AdamW Results

lr-max	lr-min	W Decay	Momentum	Val acc
0.0001	0.00001	0.0001	0.9	45.62%
0.0005	0.00005	0.0001	0.9	35.68%
0.001	0.0001	0.0001	0.9	31.08%
0.00001	0.000001	0.0001	0.9	25.32%
0.000005	0.0000005	0.0001	0.9	19.76%
0.0001	0.00001	0.00005	0.9	45.98%
0.0001	0.00001	0.0002	0.9	46.74%
0.0001	0.00001	0.00001	0.9	45.30%
0.0001	0.00001	0.0005	0.9	44.54%
0.00005	0.000005	0.0007	0.9	43.30%
0.0001	0.000005	0.0002	0.9	46.42%
0.00001	0.00001	0.0002	0.9	46.39%
0.001	0.0	0.04	0.9	41.33%
0.0002	0.0	0.04	0.9	49.78%

Figure 1. SGD: accuracy over epochs

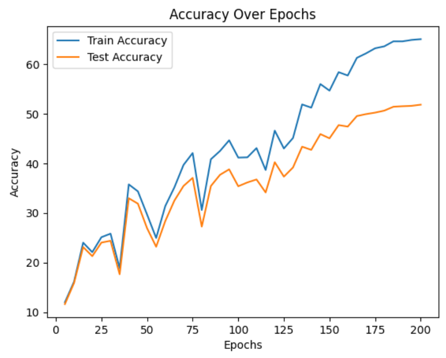
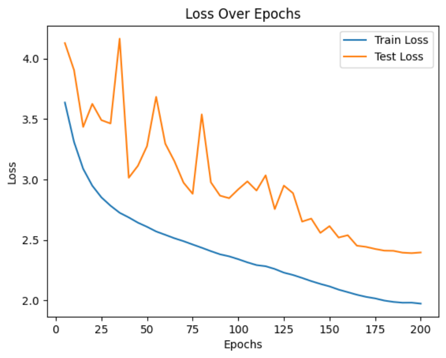


Figure 2. SGD: loss over epochs



because LARS performed for a just slightly greater batch size than SGD and worse than AdamW. Also the technological improvement of LAMB didn't help that much. This can be probably explained by the LeNet-5 network.

Figure 3. AdamW: accuracy over epochs

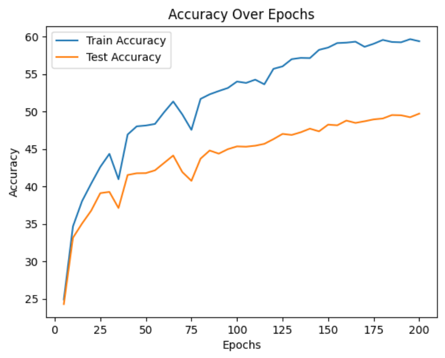


Figure 4. AdamW: loss over epochs

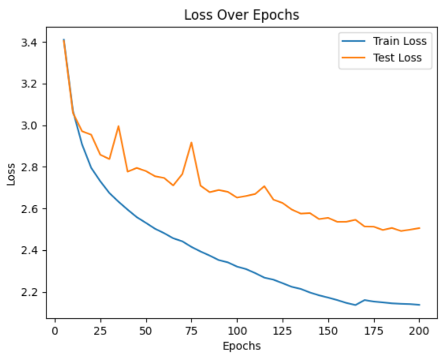


Table 4. Large Batch Results

Optimizer	Max Batch Size	Accuracy
SGD	256	49.2%
AdamW	512	47.8%
LARS	512	48.1%
LAMB	1024	46.9%

4.3. Experiment 3: Distributed Training with LocalSGD

We then modified the K number of workers and J number of local steps before synchronization. We tested $K = \{2, 4, 8\}$ and $J = \{16, 32, 64\}$ (we did not test lower values of J because it was not necessary). The results are shown in Table 5. We noticed that no relevant change has been seen using LocalSGD when compared to standard SGD, and the small differences can be explained by statistical variation. We couldn't use higher values of K and J due to the limited size of the CIFAR-100 training dataset. Except K and J, we used the same parameters of the centralized scenario for SGD. We attribute the elevated stability of localSGD compared to large batch size to the fact that the average of the

Table 5. LocalSGD Results

Configuration ($K \times J$)	Accuracy
2×16	50.1%
4×32	49.8%
8×64	49.5%

Table 6. SlowMo Results

Configuration ($K \times J$)	Accuracy
2×16	50.3%
4×32	49.9%
8×64	49.6%

weight-changes are around the correct values.

4.4. Experiment 4: Distributed Training with SlowMMO

We applied SlowMMo using the same values of K and J to see if this changes positively or negatively the performances. Table 6 shows the results. There are no significant changes.

4.5. Experiment 5: Local workers with different computational power

To run this, we pass computational-power-unequal=True. We tried to scale J up for K=4. This implied an unproportional division of the local steps: [1/15, 2/12, 4/15, 8/15] In 7 it is possible to see the results. As we expected, the results started degrading for J=16. So we applied use-weighted-avg-for-J-unequal=True to make an average weighted for the given steps before avg at each local worker. For the second part, on the centralized dataset, we multiplied the tensor of the weights by a scalar that changes with the model i: [1/3.5, 2/3.5, 4/3.5, 8/3.5]. This weighted average is applied to compute the global model from the local workers, giving more importance on the workers with a greater number of local steps. Surprisingly, this didn't help improving the accuracy. We think this may be related to the first problem we have provided: the first that some workers do an extensive number of local steps. To address this problem, we then tried with slowMMO (with and without AVG weighted based on J). We expected that a smoother general model update would fix this problem and lead to a good convergence, but also this experiment failed. We assume this could be related to an intrinsic problem related to instability in training results for both the workers that elaborate a lot of data for each step and the ones that elaborate very few. Table 7 shows the results.

Table 7. Local workers with different computational power results

K	J	Use AVG Weighted for J?	slowMMo?	Accuracy
4	8	False	no	51.28%
4	16	False	no	49.34%
4	16	True	no	49.04%
4	16	False	yes	49.52%
4	16	True	yes	48.84%

5. Limitations

By structuring the methodology into these steps, the project systematically tackled challenges like hardware limitations, scalability, and model generalization. We used a LeNet-5 architecture that does not offer great performance. In the future it would be good to try with better neural networks. We could evaluate further improvements in the learning rate. Also CIFAR-100 datasets limited our performance. We should investigate to find better solutions to improve the data in the situation where they do an unequal number of local steps.

6. Conclusion and Future Directions

This project demonstrates the potential of distributed learning for training deep neural networks efficiently. Key findings include:

- LocalSGD balances computation and communication
- LARS and LAMB improve large-batch training
- Using slowMMO doesn't significantly change the results
- Distributing unequally between the workers the number of steps before averaging causes a significant performance's drop. The techniques we used (weighted avg and slowMMO) didn't improve the performance.

We have found that large batches can be a system for scaling, but localSGD offers significantly better performance on LeNet-5. For better performance, or future directions: it needs to be changed the dataset: we suggest for future experiments to use the AlexNet dataset. Also, because LeNet-5 is a limitation too, we could try a more powerful network such as ResNet. Starting from a shorter batch size or applying weights averaging also after multiple epochs of local training. Furthermore, it is good to study systems to reduce the synchronization time. Exploring advanced neural architectures on larger datasets can be another way and also Implementing asynchronous communication to further reduce synchronization delays. Finally, we should investigate new methods in the case of different steps before averaging distributed differently between the local workers.

References

[1] Sebastian U. Stich. Don't use large mini-batches, use local sgd. In *ICLR*, 2020. 1, 2

[2] Jian Wang, Shen Li, Pengfei Wu, Zhouchen Lin, and Hongyi Li. Slowmo: Improving communication-efficient distributed sgd with slow momentum. In *ICLR*, 2020. 2

[3] Yang You, Ilya Gitman, and Boris Ginsburg. Large batch training of convolutional networks. 2017. 2

[4] Yang You, Ilya Gitman, and Boris Ginsburg. Large batch optimization for deep learning: Training bert in 76 minutes. In *ICLR*, 2020. 2, 3, 4

[5] Hu Yu, Rong Jin, Yi Yang, and Michael I. Jordan. On the linear speedup analysis of communication efficient momentum sgd for distributed non-convex optimization. 2019. 2