Learning Optimizers for Local SGD

Charles-Étienne Joseph[1,2,*]
charles-etienne.joseph@mila.quebec

Benjamin Thérien[1,2,*]
benjamin.therien@mila.quebec

Abhinav Moudgil[1,3]
abhinav.moudgil@mila.quebec

Boris Knyazev[4]
b.knyazev@samsung.com

Eugene Belilovsky[1,3]
eugene.belilovsky@concordia.ca

Abstract

Communication-efficient variants of SGD, specifically local SGD, have received a great deal of interest in recent years. These approaches compute multiple gradient steps locally, that is on each worker, before averaging model parameters, helping relieve the critical communication bottleneck in distributed deep learning training. Although many variants of these approaches have been proposed, they can sometimes lag behind state-of-the-art optimizers for deep learning. In this work, we incorporate local optimizers that compute multiple updates into a learned optimization framework, allowing to meta-learn potentially more efficient local SGD algorithms. Our results demonstrate that local learned optimizers can substantially outperform local SGD and its sophisticated variants while maintaining their communication efficiency. We show that the learned optimizers can generalize to new datasets and architectures, demonstrating the potential of learned optimizers for improving communication-efficient distributed learning.

1 Introduction

Rapidly training large-scale deep learning models is a problem of continued interest in the community. However, the communication overhead associated with distributed optimization can often lead to inefficient use of computing resources and increased wall clock times (Lin et al., 2018). The primary communication overhead of distributed SGD comes from the synchronization of gradients computed by different workers. A recently popular direction to alleviate this overhead is local SGD (Stich, 2019), where each worker computes multiple gradient steps independently before aggregating the weights of their local models. This reduces the communication costs.

Local SGD, however, has a number of challenges limiting its practical use: 1) models tend to diverge after many local steps (Wang et al., 2019); 2) local SGD can be challenging to combine with state-of-the-art adaptive optimization strategies needed to train certain model classes (e.g., transformers (Vaswani et al., 2017)); and 3) local SGD introduces a complex dynamic between the local and global updates, which can lead to complex interactions between global and local learning rates (Reddi et al., 2020).

Learned optimization through meta-learning has been an increasingly important topic of research interest (Andrychowicz et al., 2016). Advances have been made in both scalable architectures (Wichrowska et al., 2017) and meta-learning strategies (Vicol et al., 2021). Recent works have demonstrated highly competitive performance with state-of-the-art adaptive optimization strategies (Metz et al., 2022a,b),

[1] Mila - Quebec AI Institute [2] Université de Montréal [3] Concordia University [4] Samsung - SAIT AI Lab, Montréal [*]Equal contributions

International Workshop on Federated Learning in the Age of Foundation Models in Conjunction with NeurIPS 2023 (FL@FM-NeurIPS'23). This workshop does not have official proceedings and this paper is non-archival.

showing strong meta-generalization on new architectures and datasets. This progress suggests that these approaches can potentially serve as off-the-shelf replacements for existing adaptive optimization methods.

In this work, we propose learned optimization as an approach to alleviate the challenges of communication-efficient distributed learning. We, therefore, take the first steps to investigate if learned optimization can be used to improve communication-efficient distributed learning, and in particular local SGD and its variants. Our main contributions are:

- We demonstrate that learned optimizers can be used to augment local SGD for communication-efficient distributed deep learning, outperforming strong baselines.
- We propose and evaluate two architectures for the learned optimization of local SGD.
- We demonstrate that our local learned optimizers, even when meta-learned on a single or few architecture and dataset combinations, can generalize to new architectures and datasets obtaining competitive results in communication-efficient distributed settings.

2 Methodology

Our method builds upon the local SGD framework (Stich, 2019), by learning to aggregate local model weights $\{\boldsymbol{w}_{t,h}^{(k)}\}_{k=0}^{K-1}$ during communication rounds (see Algorithm 1 in supplement B). Specifically, at each communication round t, on all K clients, we take H local steps of SGD using a local minibatch of size B_{loc} for each local step h. After H local steps, we employ a per-parameter learned optimizer F_ϕ to compute the updated centralized weights. F_ϕ receives as input the difference between the initial and final weights ($\Delta_t^{(k)}$) for each worker k and the learned optimizer state ($\boldsymbol{u}_t$); it outputs the global update.

2.1 Learned Optimizer Training and Architectures

We consider the meta-learning framework with a learned optimizer F_ϕ with parameters ϕ that is used to optimize a model with parameters $\boldsymbol{w}$. In the meta-learning formulation, ϕ is obtained by solving the following optimization problem:

$$\min_{\phi} \mathbb{E}_{(\mathcal{D},\boldsymbol{w}_0)\sim\mathcal{T}} \mathbb{E}_{(X,Y)\sim\mathcal{D}} \left(\frac{1}{T} \sum_{t=0}^{T-1} \mathcal{L}(X, Y; F_\phi(\cdot)) \right), \tag{1}$$

where $\mathcal{T}$ is a distribution over optimization tasks defined as pairs of dataset $\mathcal{D}$ and initial weights $\boldsymbol{w}_0$ associated with a particular neural architecture, ϕ represents the weights of the learned optimizer, and T is the length of the unroll which we write as a fixed quantity for simplicity. In practice, during meta-optimization, we can vary T according to a truncation schedule (Metz et al., 2022a).

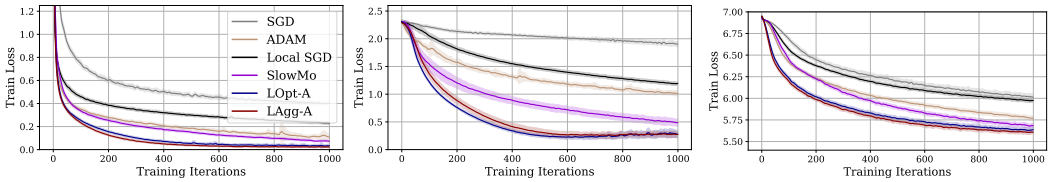
In our experiments, F_ϕ is an MLP with 2 hidden layers and 32 hidden nodes per layer. The input to F_ϕ is based on a diverse set of features computed based on Δ_t and state $\boldsymbol{u}_t$ such as different kinds of momentum analogous to adaptive optimizers and AdaFactor features (Shazeer & Stern, 2018; Metz et al., 2022a). Their computation is detailed in the supplement section D.

LOpt-A Our first proposed variant of a locally learned optimizer uses Δ_t, the average of the updates from all workers, as an input feature and uses it to compute features along with the optimizer state. This process is analogous to existing learned optimization proposed in Metz et al. (2022a) where the role of the gradient is replaced with Δ_t.

LAgg-A Our second locally learned optimizer takes advantage of pre-aggregated information from each worker, specifically it uses all the $\Delta_t^{(k)}$ as input to the MLP along with the AdaFactor features computed from Δ_t, the average of the updates from all workers: $F_\phi\left(\mathbf{A}_t, \Delta_t^{(0,1,\ldots,K-1)}\right)$. We refer to it as a *locally learned aggregator* as it learns to aggregate the weights updates coming from K workers. This variant generalizes our LOpt-A and is potentially more powerful, however, we found that LOpt-A can also perform well while being simpler.

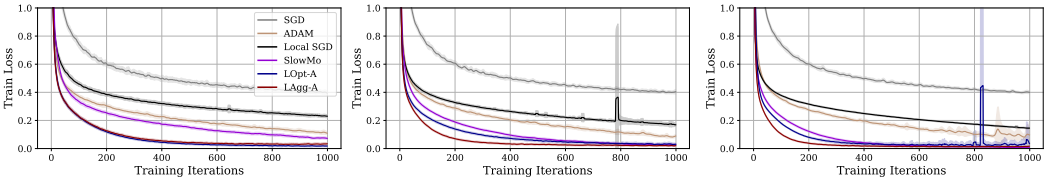
3 Experiments

Our empirical evaluation is based on standard supervised learning tasks with different dataset and architecture combinations commonly studied in learned optimization literature Metz et al. (2022a). For each task, we use a local batch size B_{loc} of 128, while the rest of the configuration varies depending on the experiment. These configurations along with training hyperparameters, baseline



(a) FMNIST (b) Cifar-10 (c) ImageNet

Figure 1: **Learned optimizers enable communication-efficient learning.** Our LOpt-A and LAgg-A outperform strong communication-efficient baselines such as SlowMo and local SGD. They also outperform well tuned standard optimization strategies at equivalent effective batch sizes.



(a) H=4 (b) H=8 (c) H=16

Figure 2: **LAgg-A outperforms all optimizers for $H \in \{4, 8, 16\}$ local steps**. All training curves are reported for the 28×28 FMNST dataset. The top row plots training curves for a small CNN, while the bottom row plots training curves for an MLP. All experiments use $K = 8$.

hyperparameters, and details of datasets and architectures used are detailed in the supplement sections E. We present several baselines to place our local learned optimizer's performance within a meaningful context. We train models using **SGD** (Robbins, 1951) and **Adam** (Kingma & Ba, 2017) to provide a comparison to non-local algorithms. At each step, these baselines compute updates using the same effective batch size $K \times H \times B_{loc}$ as the local optimizers they are compared to. We also provide two communication-efficient distributed baselines: local SGD (Stich, 2019) and SlowMo (Wang et al., 2019). An extensive hyper-parameter search is conducted for each baseline in every configuration. The best hyperparameters are reported in the supplement section F.

3.1 Evaluating LAgg-A and LOpt-A In Distribution

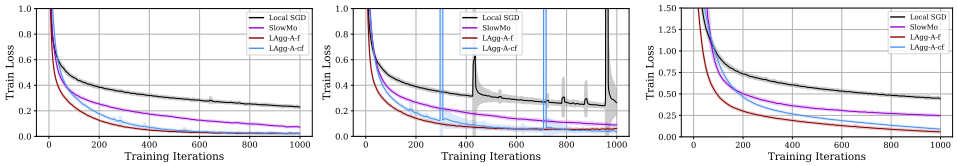
In this section, we evaluate our proposed optimizers on three datasets using $H = 4$ iterations and $K = 8$ workers. Following the evaluation protocol of Metz et al. (2022a), in each case, we meta-train on a task (dataset and architecture pair) and perform evaluation on a new seed. That is, in distribution evaluations test the generalization of the optimizer to new initialization of the model and new ordering of the data. Results on FMNIST 2-Layer MLP (left), CIFAR-10 CNN (center), and ImageNet 3-Layer MLP (right) are reported in Figure 1. We observe that our learned optimizers enjoy strong convergence, obtaining lower training loss in fewer iterations than all baseline models. Note that the SlowMo is well-tuned and represents a very competitive approach in the class of methods that perform local updates (Wang et al., 2019).

3.2 The Effect of Local Iterations (H)

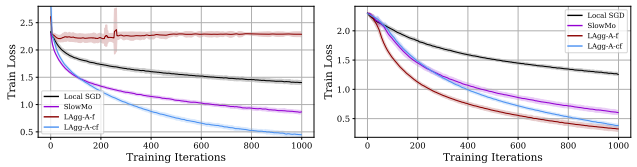
We now analyze our local learned optimizers' capability to scale to a larger number of local iterations (H). Specifically, we vary $H \in \{4, 8, 16\}$ and meta-train our learned optimizers on the FMNIST 2-Layer MLP task for each case. We also report the performance of corresponding tuned baselines with the equivalent batch size. The results are reported in Figure 2. We also show the number of communications to achieve a fixed training loss (0.2) in Table 1 of the supplement. We observe that even for relatively high H (Lin et al., 2018) there is an improvement over the strong communication-efficient baselines. As expected, Table 1 illustrates higher H yields more rapid convergence on a per training iteration basis (due to more samples being processed). We also observe that LAgg-A begins to show a substantial advantage compared to LOpt-A at this higher H value. We thus focus on this optimizer in our subsequent meta-generalization studies. A similar study of the effect of the number of workers K is provided in the supplement section C.2.

3.3 Meta-generalization

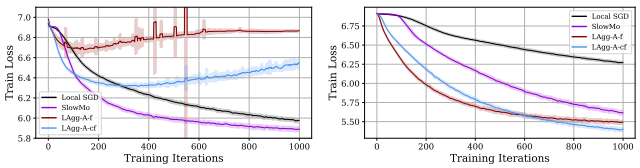
This section evaluates the meta-generalization capabilities of our local learned optimizers in communication-efficient settings. For brevity, we focus on Figure 3 here, leaving the results of Figure 5, showing generalization to different numbers of local steps H, to the supplement C.3. **LAgg-**



(a) FMNIST, 2-Layer MLP (b) FMNIST, 3-Layer MLP (c) FMNIST, CNN



(d) CIFAR-10, 2-Layer MLP (e) CIFAR-10, CNN



(f) ImageNet, 2-Layer MLP (g) ImageNet, CNN

Figure 3: **Meta-generalization to new datasets and new architectures**. All optimizers were meta-trained and hyper-parameter tuned for task (a). Meta-generalization is evaluated on new architectures same dataset (plots (b),(c)), new dataset same architecture (plots (d),(f)), and new dataset and new architecture (plots (e),(g)).

A-f is trained on the FMNIST, 2-Layer MLP task, while **LAgg-A-cf** is trained on a two-dataset task using FMNIST and CIFAR-10 with the same 2-Layer MLP. All baseline models use hyperparameters tuned on the FMNIST 2-Layer MLP task. Every model is trained using $K = 8$ and $H = 4$.

Generalization to unseen architectures We observe that our learned optimizers can generalize to unseen architectures (Figure 3 plots (b),(c)). In particular, LAgg-A-f trained on 2-Layer MLP tasks can perform well on a CNN and an MLP of different depth, highlighting the practicality of our approach. Performance in the case of the CNN is particularly strong without having observed this architecture during training

Generalization to unseen datasets We observe that LAgg-A meta-trained on FMNIST 2-Layer MLP struggles to optimize the same architecture on CIFAR-10 and Imagenet. We note, however, that including an additional task (CIFAR-10, MLP) during meta-learning can significantly improve performance. Specifically, we observe that this learned optimizer (LAgg-A-cf) is able to generalize to both of its in-distribution tasks (CIFAR-10 and FMNIST MLP) as well as improve performance on Imagenet MLP. This suggests that stronger meta-generalization can be achieved by scaling the training tasks in our communication-efficient setting as has been demonstrated for standard optimization settings in the learned optimization literature (Metz et al., 2022b).

Generalization to unseen datasets and architectures Interestingly, we observe (Figure 3 plots (e),(g)) that both learned optimizers, Lagg-A-f and LAgg-A-cf achieve strong generalization when varying both the dataset (CIFAR-10 and ImageNet) and the architecture (CNN).

4 Conclusion

We demonstrated the utility of learned optimization for improving communication-efficient distributed training of deep networks, proposing two learned optimizer architectures — LAgg-A and LOpt-A. Our results illustrate that these optimizers can effectively be applied in communication-efficient distributed settings. We highlight their generalization capabilities to unseen architectures and datasets. These findings establish learned optimization as a promising direction for improving communication-efficient distributed training algorithms for deep learning while scaling to diverse architectures, datasets, and H values. They also hold promise not only in the current context but also in decentralized and federated learning scenarios.

Acknowledgements

We acknowledge support from Mila-Samsung Research Grant and the CERC in Autonomous AI.

References

Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. *Advances in neural information processing systems*, 30, 2017.

Brandon Amos. Tutorial on amortized optimization for learning to optimize over continuous domains. *arXiv e-prints*, pp. arXiv–2202, 2022.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. *Advances in neural information processing systems*, 29, 2016.

Tianlong Chen, Weiyi Zhang, Zhou Jingyang, Shiyu Chang, Sijia Liu, Lisa Amini, and Zhangyang Wang. Training stronger baselines for learning to optimize. *Advances in Neural Information Processing Systems*, 33:7332–7343, 2020.

Tianlong Chen, Xiaohan Chen, Wuyang Chen, Zhangyang Wang, Howard Heaton, Jialin Liu, and Wotao Yin. Learning to optimize: A primer and a benchmark. *The Journal of Machine Learning Research*, 23(1):8562–8620, 2022.

James Harrison, Luke Metz, and Jascha Sohl-Dickstein. A closer look at learned optimization: Stability, robustness, and inductive biases. *Advances in Neural Information Processing Systems*, 35:3758–3773, 2022.

Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *Artificial Neural Networks—ICANN 2001: International Conference Vienna, Austria, August 21–25, 2001 Proceedings 11*, pp. 87–94. Springer, 2001.

Jinlong Ji, Xuhui Chen, Qianlong Wang, Lixing Yu, and Pan Li. Learning to learn gradient aggregation by gradient descent. In *IJCAI*, pp. 2614–2620, 2019.

Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. Scaffold: Stochastic controlled averaging for federated learning. In *International conference on machine learning*, pp. 5132–5143. PMLR, 2020.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

Xiang Li, Kaixuan Huang, Wenhao Yang, Shusen Wang, and Zhihua Zhang. On the convergence of fedavg on non-iid data. *arXiv preprint arXiv:1907.02189*, 2019.

Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. In *International Conference on Machine Learning*, pp. 3043–3052. PMLR, 2018.

Tao Lin, Sebastian U. Stich, and Martin Jaggi. Don't use large mini-batches, use local SGD. *CoRR*, abs/1808.07217, 2018. URL `http://arxiv.org/abs/1808.07217`.

Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pp. 1273–1282. PMLR, 2017.

Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, pp. 4556–4565. PMLR, 2019.

Luke Metz, Niru Maheswaranathan, C Daniel Freeman, Ben Poole, and Jascha Sohl-Dickstein. Tasks, stability, architecture, and compute: Training more effective learned optimizers, and using them to train themselves. *arXiv preprint arXiv:2009.11243*, 2020.

Luke Metz, C. Daniel Freeman, James Harrison, Niru Maheswaranathan, and Jascha Sohl-Dickstein. Practical tradeoffs between memory, compute, and performance in learned optimizers, 2022a.

Luke Metz, James Harrison, C Daniel Freeman, Amil Merchant, Lucas Beyer, James Bradbury, Naman Agrawal, Ben Poole, Igor Mordatch, Adam Roberts, et al. Velo: Training versatile learned optimizers by scaling up. *arXiv preprint arXiv:2211.09760*, 2022b.

Konstantin Mishchenko, Grigory Malinovsky, Sebastian Stich, and Peter Richtárik. Proxskip: Yes! local gradient steps provably lead to communication acceleration! finally! In *International Conference on Machine Learning*, pp. 15750–15769. PMLR, 2022.

Adel Nabli and Edouard Oyallon. Dadao: Decoupled accelerated decentralized asynchronous optimization for time-varying gossips. *arXiv preprint arXiv:2208.00779*, 2022.

Adel Nabli, Eugene Belilovsky, and Edouard Oyallon. A^2CiD^2: Accelerating asynchronous communication in decentralized deep learning. *arXiv preprint arXiv:2306.08289*, 2023.

Jose Javier Gonzalez Ortiz, Jonathan Frankle, Mike Rabbat, Ari Morcos, and Nicolas Ballas. Trade-offs of local sgd at scale: An empirical study. *arXiv preprint arXiv:2110.08133*, 2021.

Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečnỳ, Sanjiv Kumar, and H Brendan McMahan. Adaptive federated optimization. *arXiv preprint arXiv:2003.00295*, 2020.

Herbert E. Robbins. A stochastic approximation method. *Annals of Mathematical Statistics*, 22: 400–407, 1951. URL https://api.semanticscholar.org/CorpusID:16945044.

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In Jennifer G. Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4603–4611. PMLR, 2018. URL http://proceedings.mlr.press/v80/shazeer18a.html.

Shaohuai Shi, Xiaowen Chu, Ka Chun Cheung, and Simon See. Understanding top-k sparsification in distributed deep learning. *CoRR*, abs/1911.08772, 2019. URL http://arxiv.org/abs/1911.08772.

Sebastian U Stich. Local sgd converges fast and communicates little. *arXiv preprint arXiv:1805.09767*, 2018.

Sebastian U. Stich. Local sgd converges fast and communicates little, 2019.

Sebastian U. Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified SGD with memory. *CoRR*, abs/1809.07599, 2018. URL http://arxiv.org/abs/1809.07599.

Sebastian Thrun and Lorien Pratt. *Learning to learn*. Springer Science & Business Media, 2012.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Paul Vicol, Luke Metz, and Jascha Sohl-Dickstein. Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 10553–10563. PMLR, 2021. URL http://proceedings.mlr.press/v139/vicol21a.html.

Jianyu Wang, Vinayak Tantia, Nicolas Ballas, and Michael G. Rabbat. Slowmo: Improving communication-efficient distributed SGD with slow momentum. *CoRR*, abs/1910.00643, 2019. URL http://arxiv.org/abs/1910.00643.

Jue Wang, Yucheng Lu, Binhang Yuan, Beidi Chen, Percy Liang, Christopher De Sa, Christopher Re, and Ce Zhang. CocktailSGD: Fine-tuning foundation models over 500Mbps networks. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 36058–36076. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/wang23t.html.

Olga Wichrowska, Niru Maheswaranathan, Matthew W Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Nando Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. In *International conference on machine learning*, pp. 3751–3760. PMLR, 2017.

A Related Work

Local SGD and communication-efficient DL Local SGD has been analyzed in a number of works (Stich, 2018; Lin et al., 2018) which demonstrated that it both theoretically and empirically can lead to communication savings. It has also been shown that local SGD, particularly when combined with phases of regular SGD, can lead to better generalization (Lin et al., 2018), while Ortiz et al. (2021) found this generalization trend is less clear at large scale.

Wang et al. (2019) introduced the use of global or server-side momentum and showed that it can accelerate local SGD as well as a number of decentralized and asynchronous stochastic algorithms. A closely related algorithm has been proposed and extensively used in federated learning for communication efficiency (McMahan et al., 2017; Li et al., 2019). Work in this field has largely focused on addressing the heterogeneity of data across workers or clients (Karimireddy et al., 2020; Mishchenko et al., 2022). These advancements are generally achieved by hand-designed algorithmic enhancements, whereas our approach relies on more flexible and potentially more powerful learnable mechanisms that may generalize these and more complex algorithms.

Another approach to communication-efficient learning is to compress the gradients or parameters. Two popular strategies in this setting are sparsification (Stich et al., 2018; Shi et al., 2019) and quantization (Alistarh et al., 2017) of the gradient. These strategies have also been combined in Wang et al. (2023). This line of work is thus orthogonal but complementary to our proposal. Communication efficiency has also been studied in the decentralized setting (Nabli & Oyallon, 2022; Nabli et al., 2023; Lian et al., 2018). Our work focuses on the centralized training setting but the methods can also be extended to decentralized training.

Learning to Optimize (L2O) The idea of learning to learn and meta-learning has a long history (Schmidhuber, 1992; Thrun & Pratt, 2012). Many early works in this area focused on learning to efficiently acquire general knowledge or inductive bias. Hochreiter et al. (2001) proposed to use meta-learning in direct combination with gradient-based optimization to learn a separate network, which can be seen as a learned optimizer, which performs updates on another network. Andrychowicz et al. (2016) extended these ideas to a more scalable LSTM-based per-parameter architecture and demonstrated that the learned optimizer can generalize to new problems.

A large number of follow up works have improved L2O methods (Wichrowska et al., 2017; Metz et al., 2019; Chen et al., 2020; Metz et al., 2020; Harrison et al., 2022), see Chen et al. (2022); Amos (2022) for surveys. These methods introduced different types of hierarchy into the learnable optimizer while simplifying its architecture in favor of stronger predefined features to improve its efficiency. In Metz et al. (2022a), the efficiency of these methods was further analyzed in a large-scale study and a highly efficient and simple per-parameter MLP model and feature extraction approach was introduced, which we leverage in our work. However, compared to our work these have not considered a distributed setting, where learnable optimizers may significantly alleviate the communication bottleneck.

Ji et al. (2019) proposed to learn the aggregation of gradients from workers in a distributed learning framework with a recurrent network. However, the focus was on improving non-local SGD while our work focuses on the communication efficiency in settings where each worker returns a message computed from multiple update steps. Furthermore, our approach is shown to generalize to new architectures and datasets.

B Extended Methodology

Algorithm 1 provides a detailed description of our proposed algorithm. By computing the centralized update using an expressive neural network F_ϕ, our method can be seen as a generalization of existing update methods such as taking the average iterate (Stich, 2018) or computing server-side momentum updates (Wang et al., 2019).

Algorithm 1: Learned Local Optimization

Data: Number of iterations T; Number of workers K; Number of local steps H; Local learning rate γ; Initial weights $\boldsymbol{w}_{0,0}$; Initial learned optimizer state $\boldsymbol{u}_0$; Dataset $\mathcal{D}$; Loss function $\mathcal{L}$; Learned optimizer F_ϕ

for $t \in \{0, 1, \ldots, T-1\}$ **do**
  for $k \in \{0, 1, \ldots, K-1\}$ ***in parallel*** **do**
    for $h \in \{0, 1, \ldots, H-1\}$ **do**
      $X_h^{(k)}, Y_h^{(k)} \leftarrow$ GET_LOCAL_MINIBATCH($\mathcal{D}$)
      Local step: $\boldsymbol{w}_{t,h+1}^{(k)} \leftarrow \boldsymbol{w}_{t,h}^{(k)} - \gamma \nabla_{\boldsymbol{w}} \mathcal{L}\left(X_h^{(k)}, Y_h^{(k)}; \boldsymbol{w}_{t,h}^{(k)}\right)$
    Difference in weights after H local steps: $\Delta_t^{(k)} \leftarrow \boldsymbol{w}_{t,H}^{(k)} - \boldsymbol{w}_{t,0}^{(k)}$
  $\Delta_t = \frac{1}{K} \sum \Delta_t^{(k)}$
  Compute AdaFactor features and update state: $\mathbf{A}_t, \boldsymbol{u}_{t+1} = \text{ADA}(\boldsymbol{w}_{t,0}, \boldsymbol{u}_t, \Delta_t)$
  Global update: $\boldsymbol{w}_{t+1,0} \leftarrow F_\phi\left(\mathbf{A}_t, \Delta_t^{(0,1,\ldots,K-1)}\right)$

As discussed in Reddi et al. (2020) the class of local algorithms can be described with a server-side optimizer and worker-side optimizer. For example, SlowMo (Wang et al., 2019) can be interpreted as adding momentum to the server optimization. Our design of the learned optimizer architecture only parameterizes the server-side optimization making its use more practical and scalable. Specifically, standard learned optimizers have an overhead of memory and compute. The memory must store state information and intermediate activations of the learned optimizer. In the case of our learned optimizer, this overhead (Metz et al., 2022a) is only incurred at the aggregation stage. Similarly, while the computational cost of the forward pass of learned optimizers provides a substantial overhead compared to simple add and multiply operations of SGD and Adam, for the case of our learned optimizer this cost becomes small with respect to the large amount of data processed on workers during local updates.

C Extended Results

C.1 Effect of the Local Steps (H) Continued

Table 1: **Communication rounds until achieving 0.2 loss value for different optimizers at different H values.** This table was created using the curves of figure 2

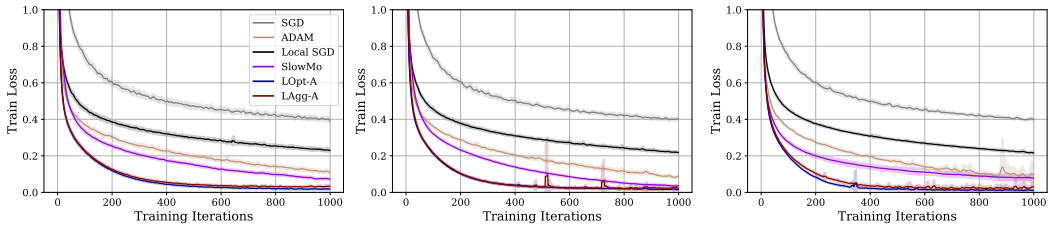
Optimizer	H=4	H=8	H=16
Local SGD	–	721	625
SlowMo	311	182	121
LOpt-A	**119**	121	89
LAgg-A	122	**81**	**55**

C.2 Effect of the Number of Workers (K)

In Figure 4 we evaluate the performance of our method as the number of workers (K) increases. Similarly to Figure 2, we vary $K \in \{8, 16, 32\}$ and meta-train our learned optimizers on the FMNIST 2-Layer MLP task for each case. We observe that our local learned optimizers gracefully scale to more workers, reaching a lower loss in fewer iterations than all baselines by a significant margin in each case.

C.3 Generalization to larger H

L-Agg-A trained for more local updates can generalize In Figure 5, we evaluate the capability of our local learned optimizers trained at one H value to generalize to another. We also evaluate whether models trained with a given number of local steps can generalize to variations in the number of local steps. Results are shown in Figure 5, here we observe that a model trained with H=16 can still perform competitively (exceeding SlowMo) when meta-evaluated with 4 local steps.



(a) K=8 (b) K=16 (c) K=32

Figure 4: **LAgg-A outperforms all optimizers for $K \in \{8, 16, 32\}$ workers**. All training curves are reported for the 2-Layer MLP with the FMNST dataset.

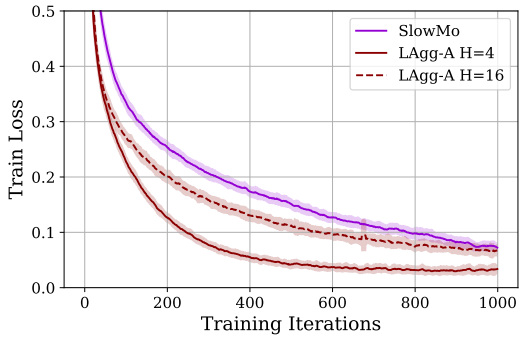


Figure 5: **LAgg-A trained at $H = 16$ generalizes to $H = 4$.** We observe that **LAgg-A H=16** trained at $H = 16, K = 8$ improves upon a strong SlowMo baseline at $H = 4, K = 8$.

C.4 Ablating Outer Loop Generalization

Following conventions in the learned optimization literature (Metz et al., 2022b,a) our focus in this work has been demonstrating the efficient convergence of the learned optimizer. Thus in our experiments, the outer loop of the meta-learning problem (Equation 1) evaluates the training data. In this section, we demonstrate that we can also obtain strong performance on the validation data using our learned optimizer. Figure 6 plots the training loss (left) and test loss (right) of our local learned optimizers trained using the validation loss objective and baselines tuned using validation loss. We observe a similar trend with respect to training loss for LAgg-A: it improves convergence across the board when compared to baseline models. On the test loss plot, LAgg-A converges significantly faster than other baselines reaching a test loss around iteration 200 that baselines only reach after 600 iterations of training. We believe this strength of LAgg-A is attributable to the meta-training objective (Equation 1) that weights the validation loss from any iteration equally, encouraging LAgg-A to immediately decrease the loss.

C.5 Ablating AdaFactor Features

Our learned optimizer leverages powerful per-parameter learned optimization features proposed in Metz et al. (2022a). Here we investigate how important these are to the performance of the optimizer. Specifically, we consider directly feeding the Δ_t or $\Delta_t^{1..K}$ to the learned optimization MLP network along with the parameter value and the 11 time features without adding any of the other momentum or AdaFactor features described in supplement D. We denote these baselines as LOpt and LAgg, respectively (excluding the -A). Results are presented in Figure 7. We observe that a large improvement in convergence and training stability is obtained by using AdaFactor features in both cases. However, we note that the performance of LOpt and LAgg alone still experiences improved convergence early in training with respect to local SGD. These baselines have no momentum calculations and the optimizer is an MLP (as opposed to a recurrent model) thus there is no way to maintain history information (unlike SlowMo's momentum). It is therefore notable that LAgg can achieve similar, albeit slower, convergence to SlowMo during the first 600 iterations. However, LAgg

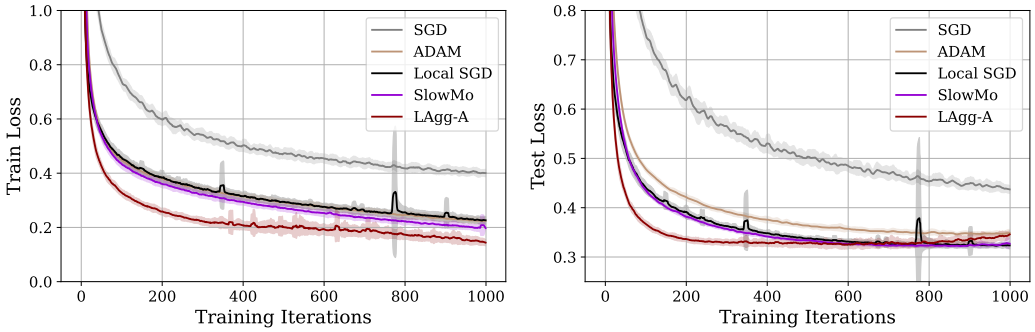


Figure 6: **Directly targeting validation loss during meta-training obtains strong performance on the test set. (Left)** plots the training loss when optimizing models on the 2-Layer MLP FMNIST task, while **(Right)** plots the validation loss. We observe that learned optimizers trained to optimize validation loss generalize in our setting. All models were trained with $K = 8$ and $H = 4$. Hand-designed optimizers were hyper-parameter-tuned to the validation set, while LAgg-A was meta-trained to optimize validation loss.

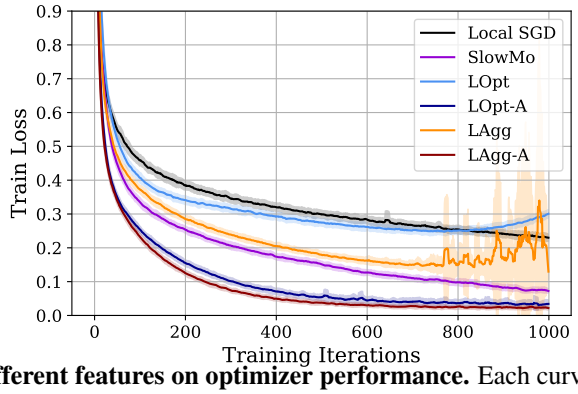


Figure 7: **Effect of different features on optimizer performance.** Each curve is an average over 10 trials with different seeds. Shaded regions represent one standard error from the mean. Each learned optimizer is trained and tested on FMNIST 2-Layer MLP at $H = 4$ and $K = 8$. Baseline models use tuned hyper-parameters reported in Table 3 of the supplement.

does seem to cause training instability from iteration 800 onwards. Interestingly, the models trained with AdaFactor features do not suffer from such instabilities, despite being trained with the same schedule as LAgg, further demonstrating their benefit.

D Learned Optimizers Architecture and Features

We use the Fashion MNIST dataset (10 classes) with full-size 28×28 images with 1 channel which we refer to as FMNIST or FMNIST 28×28. We also use the CIFAR-10 dataset (10 classes) with full-size 32×32 images with 3 channels, referred this dataset as CIFAR-10 or CIFAR-10 32×32. Finally, we use the ImageNet dataset (1000 classes) with downsampled size 32×32 images with 3 channels. We refer to this dataset as ImageNet or ImageNet 32×32.

As for neural network architectures, we use multilayer perceptron (MLP) of two different sizes, both with ReLU activations. The first has two layers of 128 hidden nodes each and we refer to it as 2-Layer MLP. The second has three hidden layers of 128 hidden nodes each and we refer to it as 3-Layer MLP. We also use a convolutional neural network (CNN) of 3 layers with ReLU activations. All 3 layers have convolution kernels of size 3×3 and use same padding. The first layer has 32 units and uses size 2 stride while the two other layers have 64 units and use size 1 stride. We refer to this architecture as CNN. The number of output values depends of the dataset with which the architecture is used.

Both our proposed learned optimizers, LOpt-A and LAgg-A, consist of a 2 hidden layer, 32 hidden nodes per layer MLP with a ReLU activation function. They share some common input features that

are detailed in Table 2 and denoted $\text{ADA}(\cdot)$ in the main text. All but the time features are normalized to have a second moment of 1 across the tensor. These features track momentum, second moments and AdaFactor row and column features stored in the inner state $\boldsymbol{u}_t$ of the learned optimizer, which are updated at each time step. Unlike prior work, all our computations are based on the average update, Δ_t. All the coefficients, β_i, are learnable parameters obtained during meta-optimization. More details of the computations for obtaining the AdaFactor row and column features can be found in Shazeer & Stern (2018).

Table 2: **Common input features of LOpt-A and LAgg-A.**

Description	
parameter value	w_t
3 momentum values with coefficients $\beta_1, \beta_2, \beta_3$	$m_{t,i} = \beta_i m_{t-1,i} + (1-\beta_i)\Delta_t$
second moment value computed from Δ_t with decay β_4	$v_t = \beta_4 v_{t-1} + (1-\beta_4)\Delta_t^2$
3 values consisting of the three momentum values normalized by the square root of the second moment	$\frac{m_{t,i}}{\sqrt{v}}$
the reciprocal square root of the second moment value	$\frac{1}{\sqrt{v}}$
3 Δ_t AdaFactor normalized values	$\Delta_t \times \text{ROW FACTOR} \times \text{COLUMN FACTOR}$
3 tiled AdaFactor row features with coefficients $\beta_5, \beta_6, \beta_7$, computed from Δ_t	$r_{t,i} = \beta_i r_{t-1,i} + (1-\beta_i)\text{ROW\_MEAN}(\Delta_t^2)$
3 tiled AdaFactor column feature with coefficients $\beta_5, \beta_6, \beta_7$ computed from Δ_t	$c_{t,i} = \beta_i c_{t-1,i} + (1-\beta_i)\text{COL\_MEAN}(\Delta_t^2)$
the reciprocal square root of the previous 6 features	$\frac{1}{\sqrt{r_{t,i} \text{ OR } c_{t,i}}}$
3 m AdaFactor normalized values	$m_{t,i} \times \text{ROW FACTOR} \times \text{COLUMN FACTOR}$
11 time features computed from the current timestep t and $x \in \{1, 3, 10, 30, 100, 300, 1000, 3000, 10\text{k}, 30\text{k}, 100\text{k}\}$	$\tanh\left(\frac{t}{x}\right)$

The state tracked by the learned optimizer thus includes

$$\boldsymbol{u}_t = \{m_{t,1}, m_{t,2}, m_{t,3}, v_t, r_{t,1}, r_{t,2}, r_{t,3}, c_{t,1}, c_{t,2}, c_{t,3}, t\}$$

Note that the AdaFactor row features are computed on a per-tensor basis. Specifically, the ROW_MEAN and COL_MEAN operation is applied on a per tensor basis. For each tensor, the corresponding components of Δ_t^2 are reshaped and their row and column means are computed. For more details please see Shazeer & Stern (2018).

Our first learned optimizer, LOpt-A, has another input feature, Δ_t, the average of all $\Delta_t^{(k)}$ coming from the K workers, for a total of 39 input features. Our second learned optimizer, LAgg-A, had K other input features which are all the different $\Delta_t^{(k)}$ coming from the K workers, for a total of $38 + K$ input features. Those features are also normalized like the others.

Both MLP output two values, a magnitude m and a scalar direction d that are use to compute the parameter update with the formula $\lambda_1 d \exp(\lambda_2 m)$ where λ_1 and λ_2 are constants values of 0.001 to keep initial step sizes small.

With all of this in mind we can compute the number of meta-parameters ϕ in the MLP for each of our learned optimizers. LOpt-A has a total of 2402 meta-parameters, while LAgg-A for values $K \in \{8, 16, 32\}$ respectively have 2626, 2882 and 3394.

E Meta-training Process

As stated in Equation 1, our meta-learning objective is the average loss over T iterations. This optimization problem usually requires long unrolls of the compute graph. We alleviate problems that can arise from long unrolls by using Persistent Evolution Strategies (PES) to compute estimates of the gradients. In our study, we use a truncation schedule that samples unroll lenghts N from a

log-uniform distribution with a minimal value of $N = 100$ and a maximum value of $N = 1000$ (the maximum value with which we evaluate our learned optimizers). The idea being that we don't always need to compute the whole inner problem each time and we can rather use information from a shorter subsequence of the problem to update the weights ϕ of our learned optimizer. Our partial unrolls used with PES have a length of 50.

For most of the learned optimizers in our study, we meta-trained for 5 000 steps. The only exceptions are the learned optimizers used in Figure 2 and the learned optimizer meta-trained for ImageNet that were meta-trained for 10 000 steps. During meta-training, we used AdamW as our optimizer with a warmup cosine decay schedule. The learning rate starts at $3\text{e}{-10}$ and warms up linearly to the peak value of $3\text{e}{-3}$. It then decays to the final value of $1\text{e}{-3}$ until the end of meta-training.

F Baselines

For every configuration in which we used the baseline optimizers, namely the architecture, the dataset and the different values of K and H, we ran an exhaustive hyperparameter sweep over the following values. For SGD and Adam, we searched over the learning rate $\alpha \in \{1, 5\text{e}{-1}, 1\text{e}{-1}, 5\text{e}{-2}, 1\text{e}{-2}, 5\text{e}{-3}, 1\text{e}{-3}, 5\text{e}{-4}, 1\text{e}{-4}, 5\text{e}{-5}, 1\text{e}{-5}\}$. For local SGD, we searched over the local learning rate $\gamma \in \{1, .5, .3, .1\}$. For SlowMo, we varied the local learning rate $\gamma \in \{1, 0.5, 0.3, 0.1\}$, the slow learning rate $\alpha \in \{1/\gamma, 5\text{e}{-1}/\gamma, 1\text{e}{-1}/\gamma, 5\text{e}{-2}/\gamma, 1\text{e}{-2}/\gamma, 5\text{e}{-3}/\gamma, 1\text{e}{-3}/\gamma, 5\text{e}{-4}/\gamma, 1\text{e}{-4}/\gamma, 5\text{e}{-5}/\gamma, 1\text{e}{-5}/\gamma\}$ and the momentum $\beta \in \{0.99, 0.95, 0.9, 0.85, 0.8, 0.75, 0.7, 0.65, 0.6, 0.55, 0.5\}$. The best hyperparameters for each configuration are regrouped in Table 3.

Table 3: **Best hyperparameters for baselines**

Configuration	**SGD** (α)	**Adam** (α)	**local SGD** (γ)	**SlowMo** (γ / α / β)
FMNIST 28×28, 2-Layer MLP, $K = 8$, $H = 4$	0.1	0.01	0.3	0.1 / 1 / 0.95
FMNIST 28×28, 2-Layer MLP, $K = 8$, $H = 8$	0.1	0.005	0.3	0.1 / 1 / 0.95
FMNIST 28×28, 2-Layer MLP, $K = 8$, $H = 16$	0.1	0.005	0.1	0.1 / 1 / 0.95
FMNIST 28×28, 2-Layer MLP, $K = 16$, $H = 4$	0.1	0.005	0.5	0.1 / 1 / 0.95
FMNIST 28×28, 2-Layer MLP, $K = 32$, $H = 4$	0.1	0.005	0.5	0.3 / 1.66 / 0.9
CIFAR-10 32×32, CNN, $K = 8$, $H = 4$	1	0.01	1	0.5 / 2 / 0.9
ImageNet 32×32, 3-Layer MLP, $K = 8$, $H = 4$	1	0.001	0.3	0.1 / 1 / 0.85