A Dynamic Weighting Strategy to Mitigate Worker Node Failure in Distributed Deep Learning

Yuesheng Xu
Computer Science & Engineering
Lehigh University
Bethlehem, USA
yux220@lehigh.edu

Arielle Carr
Computer Science & Engineering
Lehigh University
Bethlehem, USA
arg318@lehigh.edu

***Abstract*—The increasing complexity of deep learning models and the demand for processing vast amounts of data make the utilization of large-scale distributed systems for efficient training essential. These systems, however, face significant challenges such as communication overhead, hardware limitations, and node failure. This paper investigates various optimization techniques in distributed deep learning, including Elastic Averaging SGD (EASGD) and the second-order method AdaHessian. We propose a dynamic weighting strategy to mitigate the problem of straggler nodes due to failure, enhancing the performance and efficiency of the overall training process. We conduct experiments with different numbers of workers and communication periods to demonstrate improved convergence rates and test performance using our strategy.**

***Index Terms*—Optimization, Distributed deep learning, Second-order method, Large-scale machine learning**

I. Introduction

In recent years, deep learning has proven to be the most successful tool in many critical fields from computer vision to natural language processing [1]. However, the increasing complexity of deep learning models and the exponential growth in data render large-scale distributed settings essential in order to effectively manage computational loads and speed up the training process. At the same time, large-scale distributed deep learning also introduces a new set of challenges. These include hardware limitations, computational complexity, communication costs, data management, and the need to improve convergence speed, all of which are critical to optimize the efficient deployment of advanced deep learning models.

In this paper, we conduct an in-depth study of two optimization techniques, Elastic Averaging Stochastic Gradient Descent (EASGD) [2] and the second-order method AdaHessian [3]. By combining and modifying these methods, we develop a dynamic weighting strategy aimed to mitigate the impact of straggler nodes due to failures and enhance both the performance and efficiency of the distributed training process.

The remainder of this paper is organized as follows: Section II describes the problem setting and the assumptions made in our approach. Section III discusses types of parallelism in distributed deep learning. Section IV reviews related work. Section V presents our proposed method, including the data overlap and dynamic weighting strategy. Sections VI and VII describe the experimental settings and present the results, respectively. Finally, in Section VIII we provide conclusions and suggestions for future work.

II. Problem Setting

We consider a deep neural network f_θ parameterized by weight vector θ, where the network is defined as a function $f_\theta : \mathbb{R}^d \to \mathbb{R}$ mapping an input space $\mathbb{R}^d$ to an output space $\mathbb{R}$ for a general classification task. The goal is to find the optimal parameters θ^* that minimize the loss function $\mathcal{L}(x_i, y_i; \theta)$ given the dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$, where (x_i, y_i) are the inputs and corresponding labels. The optimization problem can be formally stated as:

$$\theta^* = \arg\min_\theta \mathcal{L}(x_i, y_i; \theta). \tag{1}$$

In practice, we have large datasets and complex models with millions – and potentially more – of parameters. Instead of using a full batch optimization method [4], [5], we instead perform updates on the model parameters using mini-batches [4], [5]. This is formalized as follows:

$$g_t \leftarrow \frac{1}{m} \sum_i \nabla \mathcal{L}(x_i, y_i; \theta), \tag{2}$$

$$\theta_{t+1} \leftarrow \theta_t - \eta_t g_t, \tag{3}$$

where g_t is the estimated gradient computed using the mini-batch of m samples and η_t is the learning rate at the t-th iteration.

The most commonly used optimization algorithms are first-order methods such as stochastic gradient descent (SGD) [6], Momentum [7], [8], and ADAM [9]. There are also second-order methods that impose the update rule:

$$\theta_{t+1} \leftarrow \theta_t - \eta_t D_t g_t, \tag{4}$$

where D_t is an approximation of the inverse of the Hessian matrix or any diagonal matrix that captures curvature information [10]–[12].

Our approach is based on the following assumptions:

- We consider only asynchronous distributed training procedures.
- There is a single master node and multiple worker nodes, each initially with a copy of the master's model.

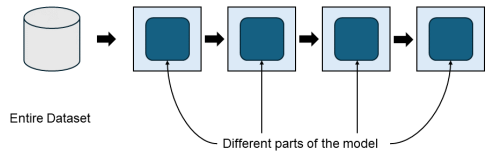

Fig. 1: This diagram illustrates the concept of **model parallelism**. Each worker holds a different segment of the model, allowing for parallel computation and handling of very large models that cannot fit into a single device.

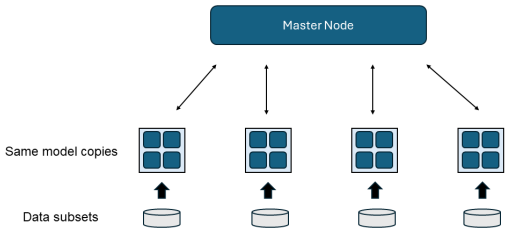

Fig. 2: This diagram illustrates the concept of **data parallelism**. Each worker node holds a subset of the dataset and communicates with the master node to update both its own model as well as the aggregated model.

- We assume a similar data distribution at each worker node.
- The communication cost between the master and worker nodes dominates the computation cost of a single worker node.

Of course, worker nodes can fail. In this paper, and operating under the assumptions listed above, we aim to handle worker node failure at the algorithmic level. Implementation of hardware level or network level mechanisms to detect the absence of a node is future work.

III. Parallelism in Distributed Deep Learning

A. Model Parallelism

Model parallelism refers to the distribution of a neural network's architecture across different computational units [13]. In this approach, different parts of the model are located on different devices, allowing for the simultaneous computation of different layers or segments of a network. See figure 1 for a visual representation.

For instance, let f_{θ_1} and f_{θ_2} be two distinct parts of a neural network model that can be trained in parallel on separate devices, where the output of f_{θ_1} serves as the input of f_{θ_2}:

$$f_\theta(x) = f_{\theta_2}(f_{\theta_1}(x)). \tag{5}$$

This can be a solution for training extremely large models that do not fit into the memory of a single device. However, this approach creates a dependency chain as subsequent computations must wait for the preceding ones to complete before proceeding.

B. Data Parallelism

In contrast to model parallelism, data parallelism splits the dataset into smaller subsets and distributes the subsets across multiple worker nodes; see figure 2. Each worker node processes its own branch of data and updates its local model parameters independently. At each iteration t, given a local subset of data $\mathcal{D}_t^i$, the local loss for worker i is computed as follows:

$$\mathcal{L}^i(\theta_t^i) = \frac{1}{|\mathcal{D}_t^i|} \sum_{(x_j, y_j) \in \mathcal{D}_t^i} \mathcal{L}(x_j, y_j; \theta_t^i), \tag{6}$$

where θ_t^i represents the model parameters at iteration t. Each worker node then communicates its local model parameters to the master node. The global parameter update at iteration t can be expressed in a more general form as:

$$\theta_{t+1} = \mathbf{Agg}(\{\theta_t^i\}_{i=1}^N), \tag{7}$$

where N is the total number of workers and $\mathbf{Agg}(\cdot)$ is a function that combines the local parameters from all workers. This step ensures that all workers contribute to the learning process and the aggregated model converges to a solution that is informed by the entire dataset.

C. Synchronous and Asynchronous Methods

In general, data-parallel distributed optimization methods are categorized into two distinct types: synchronous and asynchronous. In the synchronous approach, all worker nodes must synchronize their updates for each communication period, which can lead to bottlenecks if some nodes are slower than the others. The necessity for the master node to wait for all worker nodes makes the synchronous approach less scalable in a large-scale system [14].

On the other hand, asynchronous methods do not require all nodes to synchronize their updates with the master node. A worker node can proceed to the next task after its communication with the master node without waiting for updates from the other worker nodes. The asynchronous approach leads to a faster overall computation time, especially with some advanced scheduling to avoid communication collision among worker nodes. However, there are some pitfalls for the asynchronous method: staleness of gradients and gradient noise [14]. The elastic averaging technique can mitigate the

staleness by pulling worker and master together. Data overlap can reduce the noise both in gradients and Hessian diagonal approximation.

IV. Related Work

A. Elastic Averaging Stochastic Gradient Descent

EASGD [2] is an advanced optimization algorithm that improves the efficiency and robustness of distributed deep learning systems. At each iteration or communication period t, the update rules for one worker node i and the master node m are:

$$\theta_{t+1}^{i} = \theta_{t}^{i} - \alpha(\theta_{t}^{m} - \theta_{t}^{i}), \tag{8}$$

$$\theta_{t+1}^{m} = \theta_{t}^{m} + \alpha(\theta_{t}^{m} - \theta_{t}^{i}), \tag{9}$$

where α is a fixed "moving rate" that controls the exploration and exploitation across worker nodes. It does so by regulating how much each worker node's parameters are influenced by the global parameters and vice versa. The parameters in the master node serve as an aggregated model. The hyper-parameter α can be seen as a force pulling worker node i's parameters and master node's parameters towards each other.

One problem, however, is if some worker node fails to synchronize with the master node. For the next possible communication which the failed worker node can successfully synchronize with the master node, the outdated model from the worker node is likely to cause adverse effects on the aggregated model. To mitigate the effect from the failed worker node, we can dynamically adjust the pulling force between worker and master nodes by employing a mechanism to identify these potential bad cases. If a worker node fails, we want the aggregated model to exert a larger pulling force on the "bad" model while the "bad" model exerts a smaller pulling force on the aggregated model. In doing so, the failed worker node has less of a (negative) impact on the overall aggregated model. We will discuss the details of handling the failed worker in the Section V.

B. Second-Order Methods

An optimizer plays a significant role in the model training. In general, there are two types of commonly used optimization methods: first-order and second-order.

First-order methods primarily use the first derivatives of the loss function with respect to the model parameters. Stochastic Gradient Descent (SGD) is a well-known optimizer for its simplicity and effectiveness on a wide range of deep learning tasks. The primary advantage of first-order methods is their computational efficiency, which makes them particularly suitable for large-scale applications. They are less memory-intensive as they do not compute and store second-order information. However, their major disadvantage lies in their potentially slower convergence rates due to their reliance on local gradient information only, especially in the presence of ill-conditioned optimization landscapes or when navigating flat regions in the loss surface.

On the other hand, second-order methods use both first- and second-order information, where the second-order information is used to precondition the gradient. The main challenge associated with second-order methods stems from the computational complexity when computing the inverse of the Hessian matrix, which naively requires $O(n^3)$ operations for n parameters. In practice, however, these methods always compute some approximation of the inverse of the Hessian, reducing complexity to $O(n)$. Moreover, in distributed systems, the dominant factor often shifts from computational cost to communication overhead. The frequent exchange of parameters between worker nodes and the master node can substantially outweigh the computational costs associated with updating each worker node's model. Therefore, it is beneficial for worker nodes to take slower yet accurate steps. If the increase in computational cost does not grow exponentially with the use of second-order methods, this strategy is highly likely to effectively reduce the overall wall-clock time by minimizing communication rounds while maintaining robust convergence.

AdaHessian [3] is a second-order optimizer that uses the approximated diagonal of the Hessian matrix to adaptively adjust the learning rate. We use this optimizer as the backbone of our distributed optimization system. There are three components that allow AdaHessian to effectively utilize second-order information.

Firstly, it uses Hutchinson's method to approximate the Hessian diagonal [15]:

$$\text{diag}(\mathbf{H}) \approx \frac{1}{n}\sum_{i=1}^{n}(\mathbf{z}_i \odot (\mathbf{H}\mathbf{z}_i)),$$

where $\mathbf{H}$ is the Hessian and $\mathbf{z}_i$ is a Rademacher vector. The Hessian vector product $\mathbf{H}\mathbf{z}_i$ takes the same amount of time as one back-propagation. Secondly, to reduce the Hessian variance, the spatial averaging Hessian diagonal for each parameter is computed by taking the average around its neighbours. Finally, **AdaHessian** adjusts the learning rate adaptively, similar to the ADAM optimizer [9]. They differ in their calculation of the second moment: the gradient is replaced by the spatial averaging Hessian diagonal.

V. Method

Our method builds on the concept of asynchronous averaging of the worker and master's parameters with the use of the second-order methods to address the challenge of **parameter desynchronization** caused by straggler nodes. Due to prior failures, such nodes update their parameters less frequently than other nodes. Previous work on using data encoding to mitigate the impact of straggler nodes by embedding redundancy directly in the data has been introduced in [16].

A. Data Overlap

In distributed deep learning, particularly when using second-order optimization methods, the overlap of data across different worker nodes can be beneficial to both the performance as well as the stability of the training process. Consider the

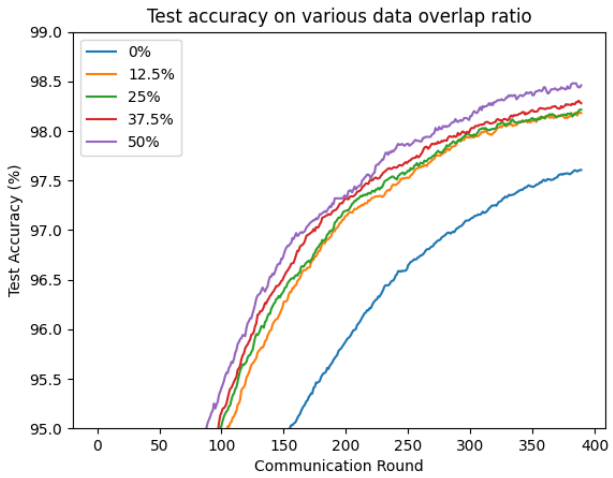

Fig. 3: Data overlap ratios: $\{0\%, 12.5\%, 25\%, 37.5\%, 50\%\}$

the set consisting of n data points $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$ and k workers. We randomly sample a of data points $\mathcal{O}$ of size $|\mathcal{O}|$. Ideally, $|\mathcal{O}|$ should be chosen carefully to balance between too much and too little overlap. For instance, if $|\mathcal{O}|$ is chosen to be too large, this diminishes the benefits of distributed system as it counteracts the purpose of speeding up the process by parallelizing the computation. Conversely, a small or no overlap can potentially lead to sub-optimal Hessian approximations. In more complex and realistic environments, the data distribution across different workers can vary significantly. Without a certain degree of data overlap, workers in general can encounter very different loss landscapes, thus generating Hessian approximations with high variance. Ongoing work focuses on addressing the performance of our method in real-world applications and scenarios.

All workers share a subset of $|\mathcal{O}| = o$ data points. The remaining data points $\mathcal{D} - \mathcal{O}$ are then randomly distributed among the k workers. Each worker w_j will receive the shared subset $\mathcal{O}$ and a unique subset $\mathcal{S}_j$ of size $|\mathcal{S}_j| = \lfloor \frac{n-o}{k} \rfloor$. Therefore, the dataset assigned to worker w_j can be characterized as:

$$\mathcal{D}_j = \mathcal{O} \cup \mathcal{S}_j,$$

where $j \in \{1, 2, \ldots, k\}$, $\bigcup_{j=1}^{k} \mathcal{S}_j = \mathcal{D} - \mathcal{O}$ and $\mathcal{S}_i \cap \mathcal{S}_j = \emptyset$ for $i \neq j$.

We provide empirical observations to justify the careful balancing of the overlap. Let $r = \frac{o}{n}$ be the ratio of the overlapped data shown by all k worker nodes. As shown in figure 3, we compare different data overlap ratio on **EAHES**. As expected, we observe a positive relationship between the data overlap ratio and test accuracy.

B. Dynamic Weight

In order to mitigate the influence of a failed worker on the master node, we propose a dynamic weighting strategy. Intuitively, we want to (1) reduce the bad influence from the failed node by increasing α in equation (8), and (2) correct the failed node using the aggregated model by decreasing α in equation (9). Instead of a fixed α value, we map a raw score that measures the change in distance between a worker and the master node to the dynamic weight. We choose piece-wise linear functions based on the observation that if a worker fails, its raw score becomes negative in the next few time steps. If a worker node works properly, its raw score usually is small and positive, and we approximate the behavior of EASGD in this case. For a worker node w_i, we compute the 2-norm of its model θ^i and the estimate of the master model $\tilde{\theta}^m$. In practice, we can acquire this estimation from other workers efficiently since communication among workers is much faster. We further define a term to estimate how far away worker model w_i is from the aggregated model:

$$u_t^i = \log(||\theta_t^i - \tilde{\theta}_t^m||).$$

We store the p most recent u^i – that is, $\{u_{t-p-1}^i, ..., u_{t-2}^i, u_{t-1}^i, u_t^i\}$ – and compute a weighted raw score a for worker i at time step t as follows:

$$a_t^i = c_{p-1}(u_{t-p-1}^i - u_{t-p-2}^i) + ... + c_1(u_{t-1}^i - u_{t-2}^i) \quad (10)$$
$$+ c_0(u_t^i - u_{t-1}^i), \quad (11)$$

where $c_{p-1} + ... + c_2 + c_1 + c_0 = 1$. Preferably, we want to apply larger weights on the most recent terms. Then, we apply a piece-wise linear mapping to obtain our weight as described next.

For simplicity, we drop the super and subscripts for a as given in equation (10). The following pair of h_1, h_2 are one choice, though we note that we do not have to limit these to linear functions. Let $h_1(a)$ be a piece-wise linear function defined as follows:

$$h_1(a) = \begin{cases} 1 & \text{if } a < k, \\ 1 + \frac{1-\alpha}{k}(a - k) & \text{if } k \leq a \leq 0, \\ \alpha & \text{if } 0 < a, \end{cases}$$

for some constant $k < 0$. Similarly, we define $h_2(a)$ as:

$$h_2(a) = \begin{cases} 0 & \text{if } a < k, \\ -\frac{\alpha}{k}a + \alpha & \text{if } k \leq a \leq 0, \\ \alpha & \text{if } 0 < a. \end{cases}$$

We replace equations (8) and (9) with

$$\theta_{t+1}^i = \theta_t^i - h_1(a)(\theta_t^i - \theta_t^m), \quad (12)$$

$$\theta_{t+1}^m = \theta_t^m + h_2(a)(\theta_t^i - \theta_t^m). \quad (13)$$

VI. Experiment Settings

We run our experiments on MNIST dataset [17] with $k \in \{4, 8\}$ local workers and the communication period[1] $\tau \in \{1, 2, 4\}$. Our experiments are conducted on a single device to simulate a master-worker distributed system due to a hardware limitation. For all methods, we compare training

[1]The communication period τ controls how often a worker node communicates with the master node. For example, if $\tau = 2$, every 2 iterations, a worker will communicate with the master node.

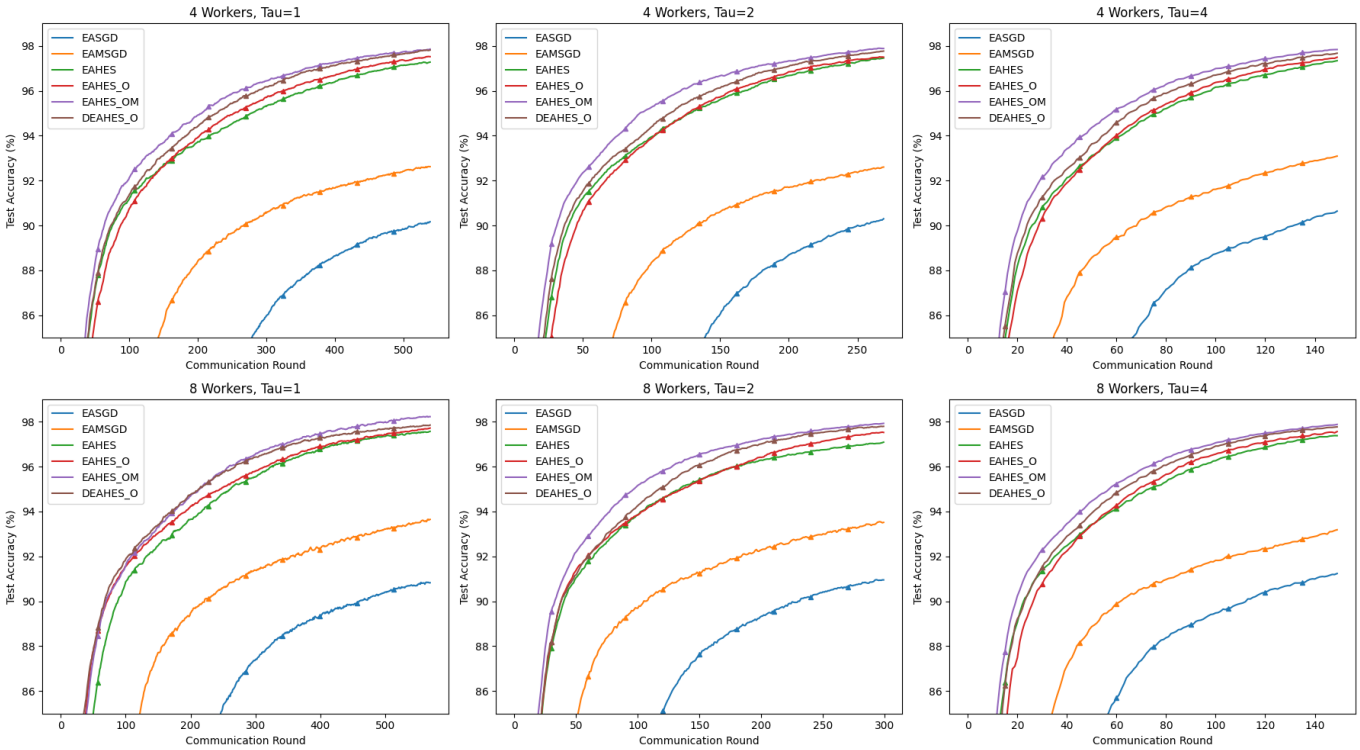

Fig. 4: Test accuracy over communication rounds for workers $k \in \{4, 8\}$ and communication period $\tau \in \{1, 2, 4\}$. Each experiment is averaged over 3 runs.

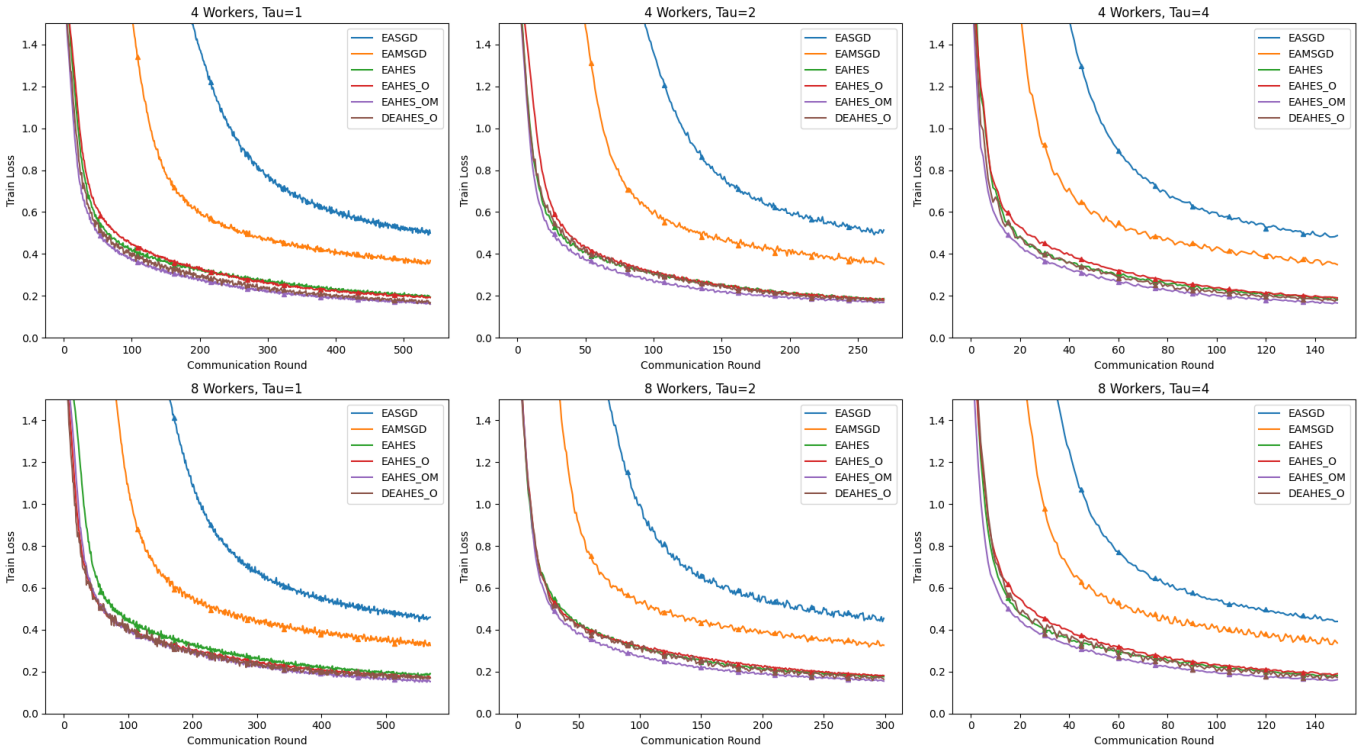

Fig. 5: Training loss over communication rounds for workers $k \in \{4, 8\}$ and communication period $\tau \in \{1, 2, 4\}$. Each experiment is averaged over 3 runs.

loss and test accuracy after a full communication round and we suppress the communication between a worker node and the master node one-third of time. We use a simple 2-layer convolutional neural network from PyTorch. We compare the following methods in the results section:

- **EASGD**: asynchronous version of **EASGD**;
- **EAMSGD**: **EASGD** with **Momentum**;
- **EAHES**: asynchronous elastic averaging **AdaHessian**;
- **EAHES-O**: **EAHES** with data overlap (denoted as **O**);
- **EAHES-OM**: **EAHES-O** but we manually adjust the weight α as if we know when a node will fail; and
- **DEAHES-O**: **EAHES-O** with dynamic weighting.

In most cases, it is difficult to anticipate when and which worker node will fail, However, in our experiments we will make this assumption for comparative purposes.

VII. Results

For basic **EASGD**, **EAHES** and **EAHES-O**, we conduct a grid search on the fixed weight α and present the result with the best choice $\alpha = 0.1$. For the data overlap approaches, we choose the ratio $r = 12.5\%$ with 8 workers and $r = 25\%$ with 4 workers. For **EAHES-OD**, our algorithm adjusts the weight α dynamically based on the change in estimated model discrepancy. The parameters for SGD-based methods are learning rate $\eta = 0.01$ and momentum $\delta = 0.5$. The parameters for AdaHessian are learning rate $\eta = 0.01$ and $\beta = (0.9, 0.999)$, and the number of sampling for Hutchinson's method is 1. We explore different choices of communication period $\tau \in \{1, 2, 4\}$.

In figures 4 and 5, due to the precise curvature information utilized by second-order methods, AdaHessian-based approaches significantly outperform those based on SGD. We observe that **EAHES-OM** has the best performance since we know when a worker will fail and can respond accordingly. **DEAHES-O** has close performance as **EAHES-OM** and outperforms all other methods. **DEAHES-O** benefits from the dynamic weight adjustment to mitigate the bad influence on the aggregated model from some potential failure without implementing extra low-level mechanisms to detect node failure. **EAHES-O** shows a better performance than **EAHES** indicating the positive effect of the data overlap strategy for methods associated with Hessian approximation. We also observe that as we increase the number of workers from 4 to 8 and increase communication period from 1 to 2 to 4, the performance does not degrade.

VIII. Conclusions and Future Work

In this paper, we investigate the issue of failed worker nodes in distributed deep learning and their effect on the convergence of the optimizer. We combine an elastic averaging technique with the AdaHessian optimizer and explore the effectiveness of data overlap and dynamic weighting techniques to address challenges under unfavorable conditions. Our experiments demonstrate that the proposed method significantly improves robustness and performance, effectively mitigating the impact of worker failures and optimizing training efficiency in distributed deep learning environments.

For the current report, performance under wall-clock time is not yet available. Communication rounds might not reflect the true wall-clock time due to contention among workers. Indeed, more workers will inevitably suffer from diminishing marginal utility. Our future work involves reporting results after conducting experiments on a realistic distributed system. We also aim to test our approach on practical applications employing distributed deep learning such as [18].

References

[1] J. Chai, H. Zeng, A. Li, and E. W. Ngai, "Deep learning in computer vision: A critical review of emerging techniques and application scenarios," *Machine Learning with Applications*, vol. 6, p. 100134, 2021. [Online]. Available: https://www.journals.elsevier.com/machine-learning-with-applications

[2] S. Zhang, A. Choromanska, and Y. LeCun, "Deep learning with elastic averaging sgd," in *Advances in neural information processing systems*, 2015, pp. 685–693.

[3] Z. Yao, A. Gholami, X. Lei, S. Shen, K. Keutzer, and G. Biros, "Adahessian: An adaptive second order optimizer for machine learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 12, pp. 10 665–10 673, 2021.

[4] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[5] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[6] L. Bottou, "Online algorithms and stochastic approximations," in *Online Learning and Neural Networks*. Cambridge University Press, 1998.

[7] Y. Nesterov, "Smooth minimization of non-smooth functions," *Mathematical Programming*, vol. 103, no. 1, pp. 127–152, 2005.

[8] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on Machine Learning (ICML)*, 2013.

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014, published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[10] C. G. Broyden, R. Fletcher, D. Goldfarb, and D. F. Shanno, "A class of methods for solving nonlinear simultaneous equations," *Mathematics of computation*, vol. 24, no. 111, pp. 889–890, 1970.

[11] J. Nocedal and S. J. Wright, *Numerical Optimization*. Springer Science & Business Media, 2006.

[12] J. E. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM, 1996.

[13] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[14] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.

[15] C. Bekas, E. Kokiopoulou, and Y. Saad, "An estimator for the diagonal of a matrix," *Applied Numerical Mathematics*, vol. 57, no. 11-12, pp. 1214–1229, 2007.

[16] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, "Straggler mitigation in distributed optimization through data encoding," in *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, Long Beach, CA, USA, 2017, pp. 5434–5442. [Online]. Available: https://papers.nips.cc/paper/2017/hash/e6888a29fb22bbf4c67a6c3409f4a836-Abstract.html

[17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[18] D. Millard, A. Carr, and S. Gaudreault, "Deep learning for koopman operator estimation in idealized atmospheric dynamics," *arXiv preprint arXiv:2409.06522*, 2024.