Model-Distributed DNN Training for Memory-Constrained Edge Computing Devices

Pengzhen Li*, Hulya Seferoglu*, Venkat R. Dasari† and Erdem Koyuncu*
pli33@uic.edu, hulya@uic.edu, venkateswara.r.dasari.civ@mail.mil, ekoyuncu@uic.edu
*University of Illinois at Chicago, †US Army Research Laboratory

***Abstract*—We consider a model-distributed learning framework in which layers of a deep learning model is distributed across multiple workers. To achieve consistent gradient updates during the training phase, model-distributed learning requires the storage of multiple versions of the layer parameters at every worker. In this paper, we design `mcPipe` to reduce the memory cost of model-distributed learning, which is crucial in memory-constrained edge computing devices. `mcPipe` uses an on-demand weight updating policy, which reduces the amount of weights that should be stored at workers. We analyze the memory cost of `mcPipe` and demonstrate its superior performance as compared to existing model-distributed learning mechanisms. We implement `mcPipe` in a real testbed and show that it improves the memory cost without hurting converge rate and computation cost.**

***Index Terms*—deep neural networks (DNN), distributed training, edge computing devices, memory.**

I. Introduction

Massive amount of data is generated at edge networks with the emerging self-driving cars, drones, robots, wireless sensors, health monitoring devices. This vast data is expected to be processed in real-time in many time sensitive applications, which is extremely challenging if not impossible with existing centralized cloud. Indeed, transmitting such massive data to the centralized cloud, and expecting timely processing are not realistic with limited bandwidth between an edge network and centralized cloud. In many scenarios, such data cannot be processed locally on computationally- and memory-limited end-devices, and calls for distributed data processing.

Training deep neural networks (DNNs) is well-known to be a very demanding task, typically requiring vast computation, memory, and power resources. In this context, distributed DNN training enables the resource burden to be shared by multiple devices instead of being undertaken by only one device. Distributed DNN training in edge systems has unique challenges as compared to multi-GPU platforms due to constrained and heterogeneous resources. In fact, DNN training mechanisms that require large memory footprint will not be feasible for many edge computing devices.

The two main approaches to distributed DNN training differ with regards to whether the training data or the DNN model is distributed across the multiple devices. In fact, in the so-called *data-distributed* learning, each worker device trains the entire DNN with the same initial weights, but over different subsets of the training data. The weight updates are then synchronized to have a new coherent set of weights to be utilized at all devices. This approach is particularly useful when inter-device communication has little to no cost relative to computation costs, such as in wired multi-GPU platforms.

When the inter-device communication rates are low, such as in wireless edge devices, passing the huge model parameters across devices becomes a major bottleneck in data-distributed learning. A more appropriate DNN training paradigm over the edge is then *model-distributed* learning, an emerging technique where each device is only responsible with training only a certain subset of layers of the DNN. Model-distributed learning is promising to reduce the communication requirements as the full model parameters do not have to be exchanged among worker devices. Moreover, since each device keeps track of a subset of layers, it can generally promise lower storage and memory requirements as compared to data-distributed learning. Benefits of model-distributed learning over data-distributed learning over multi-GPU platforms have been investigated.

A challenge of model-distributed learning is that it requires the storage of multiple versions of the layer weights during the training phase. This ensures that the forward and backward propagation passes corresponding to one training batch always undergo the same version of the weight at a given layer or device. Unfortunately, the number of versions to be stored should be proportional to the number of workers/devices in the system, meaning that model-distributed learning may introduce high storage requirements, which puts a strain on memory constrained edge computing devices.

In this paper, we focus on model-distributed DNN training at memory-constrained edge computing devices. We design `mcPipe` to reduce the memory cost of model-distributed learning. `mcPipe` uses an on-demand weight updating policy, which reduces the amount of weights that should be stored at workers. In particular, we design a mechanism to store activation and error vectors in `mcPipe` instead of weight matrices, which significantly reduces the storage requirements. We analyze the memory cost of `mcPipe` and demonstrate its superior performance as compared to existing model-distributed learning mechanisms. We implement `mcPipe` in a real testbed and show that it improves the memory cost without hurting converge rate and computation cost as compared to baselines; data-distributed training and PipeDream [1] for VGG16.

The structure of the rest of this paper is as follows. Section II puts our work into perspective. Section III presents our system model. Section IV provides background on model distributed learning and pipelining, and introduces `mcPipe` with performance analysis. Section V provides experimental evaluation of `mcPipe` in real devices. Section VI concludes the paper.

II. Related Work

Earlier studies on model distributed learning have focused on multi-GPU platforms. In particular, GPipe [2] allows different subsequences of layers on separate workers to achieve close-to-linear speed up in DNN training. PipeDream [1] eliminates idle workers and thus improves efficiency via a dedicated scheduling and batching mechanism. Weight prediction can be used to improve the performance of both GPipe and PipeDream [3]. In [4], we design resilient model-distributed training schemes that are robust to failing or severely straggling workers. As compared to this line of work, `mcPipe` focuses on reducing the memory footprint of model-distributed DNN training.

Federated learning mechanisms train DNN models distributively across workers without exchanging data [5]–[8] as workers collect data themselves. Our setup is different as master collects the data, but our approach is complementary to federated learning mechanisms that possibly use model-distributed learning. Our work is also complementary to resource allocation mechanisms that accelerate convergence speed of [9]–[12].

We note that there are various techniques to reduce the inter-worker communication costs in DNN training such as quantization [13], [14], truncation [15], [16], gradient sparsification [17], [18], conditional execution [19], [20]. Our work is complementary to these techniques in the sense that communication among workers can be further reduced using one or more of these methods.

III. System Setup

Learning Model: The training dataset is represented by I, the label vector is $\mathbf{y}$, and the weight matrix is $\mathbf{w}$. The weight matrix of a DNN is obtained by minimizing the loss function $f(\mathbf{x})$, which is determined by the learning model, via gradient descent [21]; $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta\nabla(f(\mathbf{w}^{(t)}))$, where $\mathbf{w}(t)$ is the weight matrix at the t^{th} iteration, η is the learning rate, $\nabla(f(\mathbf{w}^{(t)}))$ is the gradient of $f(\mathbf{w}^{(t)})$.

Setup: We consider an edge computing setup where end devices, edge servers, and cloud (if available) are used for distributed DNN training. The available computing devices can be classified as master or worker devices, where master devices would like to perform DNN training on their collected data, and worker devices are willing to dedicate some of their computing resources. It is natural to have multiple masters and workers in our setup, and the roles of their devices may overlap, i.e., a device can be both a master and/or worker.

For the ease of presentation, we focus on a master/worker setup, where a master device offloads its computationally intensive tasks to Worker $n \in \mathcal{N}$ (where $\mathcal{N} \triangleq \{1, \dots, N\}$). Workers are connected to the master device as well as other workers via Ethernet or Wi-Fi Direct links, depending on their availability.

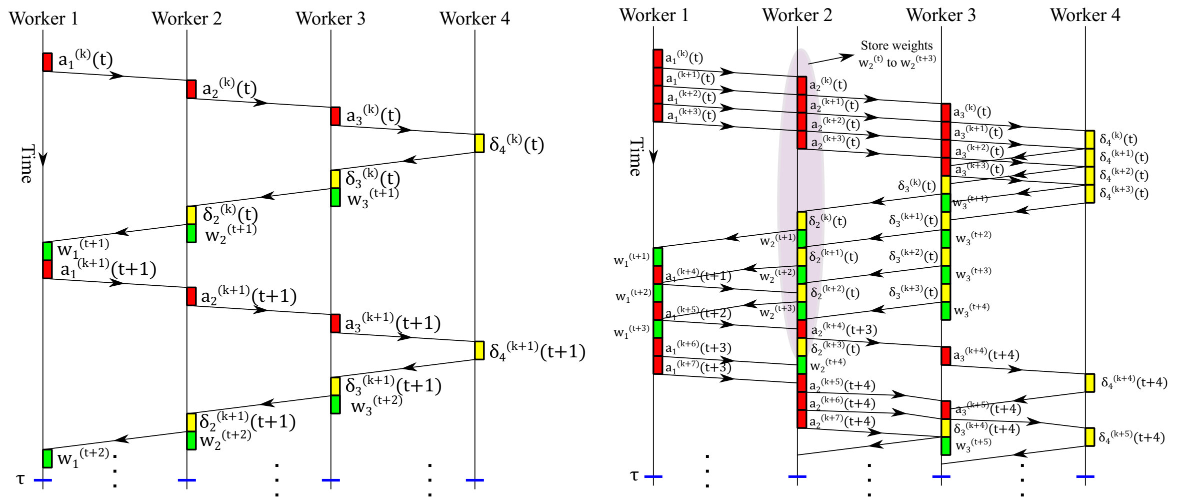
The workers have the following properties: *(i) Failures:* Workers may fail or "sleep/die" or leave the network before finishing their assigned computational tasks. *(ii) Stragglers:* Workers incur probabilistic delays in responding to the master, where delay has two components; transmission delay for exchanging data, model, and parameters, and computation delay.

IV. Memory-Constrained Model Distribution

A. Model-Distributed Learning

Model distributed learning partitions model $\mathbf{w}$, and offloads each partition to a worker. Workers collectively and distributively train the model. Each worker stores and trains a part of the model. In particular, each worker calculates activation and error vectors in the forward and back-propagation phases, respectively. In this setup, activation and error vectors instead of the whole model is exchanged among workers, so the communication cost reduces. The next example illustrates the main idea behind model distributed learning.

Example 1: We consider that the training dataset $\{I_{(1)}, \dots I_{(m)}\}$, $I_{(k)} \in \mathbb{R}^{d\times 1}$, $k \in \{1, \dots, m\}$ is used to train a 4-layer fully connected neural network, which has K neurons in each layer and the last layer is the output



(a) No Pipelining (b) Pipelining

Fig. 1: Model-distributed learning with and without pipelining. Red, yellow, and green boxes represent the computing times of activation vectors, error vectors, and weight updates, respectively.

layer. Assuming that there are four workers in the system, they collectively train the model $\mathbf{w}^{(t)} = \{\mathbf{w}_l^{(t)}\}_{l=1}^{3}$ at iteration t, where $\mathbf{w}_1^{(t)}$ is the $K \times d$ weight matrix corresponding to the first layer, while $\mathbf{w}_l^{(t)}$ is the lth layer $K \times K$ weight matrix, $l \in \{2, 3\}$.

Worker 1 receives data $I_{(k)}$ from the master device (or already has the data) at the tth iteration of weight updates ($k = t$ in this setup), and calculates $a_1^{(k)}(t) = \mathbf{w}_1^{(t)} I_{(k)}$, where $a_1^{(k)}(t) \in \mathbb{R}^K$ is the activation vector in the first layer. Worker 1 transmits $a_1^{(k)}(t)$ to Worker 2. Similarly Worker 2 and Worker 3 calculate $a_n^{(k)}(t) = \mathbf{w}_n^{(t)} g(a_{n-1}^{(k)}(t))$, where $n \in \{1, 2\}$, and g is an activation function. Worker 2 sends $a_2^{(k)}(t)$ to Worker 3, and Worker 3 sends $a_3^{(k)}(t)$ to Worker 4. This completes the forward propagation.

Worker 4 calculates its error vector $\delta_4^{(k)}(t) = g(a_3^{(k)}(t)) - y^{(k)}$, and sends $\delta_4^{(k)}(t)$ to Worker 3. The error vectors of Worker 2 and 3 are calculated as $\delta_n^{(k)}(t) = (\mathbf{w}_n^{(t)})^T \delta_{n+1}^{(k)}(t) .* g'(a_{n-1}^{(k)}(t))$ for $n \in \{2, 3\}$, where $.*$ is an element-wise multiplication, and g' is the derivative of g. Worker 3 sends $\delta_3^{(k)}(t)$ to Worker 2, and Worker 2 sends $\delta_2^{(k)}(t)$ to Worker 1. After receiving the error vectors, workers update their weight matrices. In particular, Worker 1 calculates its weight matrix according to $\mathbf{w}_1^{(t+1)} = \mathbf{w}_1^{(t)} - \eta(\delta_2^{(k)}(t) I_{(k)})$. Workers 2 and 3 update their weight vectors according to $\mathbf{w}_n^{(t+1)} = \mathbf{w}_n^{(t)} - \eta(\delta_{n+1}^{(k)}(t) g(a_{n-1}^{(k)}(t))^T)$, $n \in \{2, 3\}$. This completes the back-propagation phase. □

As seen, the model is updated across the workers in a distributive manner by only exchanging activation and error vectors among workers. The procedure of Example 1 is summarized in Fig. 1(a). Model-distributed learning is promising to reduce the communication cost as compared to data-distributed learning as the full model parameters do not have to be exchanged among worker devices; exchanging activation and error vectors is sufficient. Moreover, since each worker keeps track of a subset of layers, it can generally promise lower storage and memory requirements as compared to data-distributed learning.

B. *Pipelining for Model Distributed Learning*

Model distributed learning explained in Section IV-A is not work conserving; workers stay idle while waiting for receiving activation and error vectors from other workers, so it cannot fully utilize available computing powers at workers. For example, Worker 1 in Example 1 stays idle after transmitting $a_1^{(k)}(t)$ and receiving $\delta_2^{(k)}(t)$. Pipelining is a promising solution to address this problem by keeping workers busy. In particular, workers process multiple instances of data, activation, and error vectors. An analogy of pipelining for model distributed learning is sliding window-based transmission instead of stop-and-wait, which is similar to model distributed learning without pipelining. An example pipelining procedure based on PipeDream [1] is demonstrated in the next example and via Fig. 1(b).

Example 2: Consider the same setup in Example 1. Assume that the last weight update at each worker is the tth update, i.e., the most recent weight is $w_n^{(t)}$ at Worker n, for $n \in \{1, 2, 3\}$. Then, each worker processes up to four activation vectors before a new weight update occurs, i.e., sliding window size or mini batch size (as referred in [1]) is four. Then, it stops and waits for a weight update, and a new activation vector is calculated and sent to Worker 2 after each weight update. For example, Worker 1 calculates $a_1^{(k+p)}(t) = \mathbf{w}_1^{(t)} I_{(k+p)}$, $p \in \{0, \ldots, 3\}$ and sends them to Worker 2. Then, it stops and waits for a weight update. After weight update $\mathbf{w}_1^{(t+1)}$ is done, a new activation vector $a_1^{(k+4)}(t+1)$ is calculated and transmitted to Worker 2. Activation vectors in Worker 1 is calculated according to $a_1^{(\alpha)}(\beta) = \mathbf{w}_1^{(\beta)} I_{(\alpha)}$. Similar sliding window mechanism is used in Worker 2 and 3 and activation vectors are calculated as $a_n^{(\alpha)}(\beta) = \mathbf{w}_n^{(\beta)} g(a_{n-1}^{(\alpha)}(\beta))$, $n \in \{2, 3\}$. This completes the forward propagation phase.

Worker 4 calculates the error vector $\delta_4^{(\alpha)}(\beta) = g(a_3^{(\alpha)}(\beta)) - y^{(\beta)}$ and sends to Worker 3. The error vectors at Worker 2 and 3 are calculated as $\delta_n^{(\alpha)}(\beta) = (\mathbf{w}_n^{(\beta)})^T \delta_{n+1}^{(\alpha)}(\beta). * g'(a_{n-1}^{(\alpha)}(\beta))$, $n \in \{2, 3\}$. Workers update their weights after receiving an error vector from a neighboring worker. For example, Worker 1 calculates $\mathbf{w}_1^{(t+2)} = \mathbf{w}_1^{(t+1)} - \eta(\delta_2^{(k+1)}(t) I_{(k+1)}^T)$ after receiving $\delta_2^{(k+1)}(t)$. The general rule is that the most recent weight $\mathbf{w}_1^{(\gamma)}$ is updated using the last received error vector $\delta_2^{(\alpha)}(\beta)$ as $\mathbf{w}_1^{(\gamma+1)} = \mathbf{w}_1^{(\gamma)} - \eta(\delta_2^{(\alpha)}(\beta) I_{(\alpha)}^T)$. Similarly, Worker 2 and 3 update the most recent weight $\mathbf{w}_n^{(\gamma)}$ after receiving an error vector $\delta_{n+1}^{(\alpha)}(\beta)$ as $\mathbf{w}_n^{(\gamma+1)} = \mathbf{w}_n^{(\gamma)} - \eta(\delta_{n+1}^{(\alpha)}(\beta) g(a_{n-1}^{(\alpha)}(\beta))^T)$, for $n \in \{2, 3\}$. This completes the back-propagation phase. □

Example 2 demonstrates that workers are kept busier in pipelining, so more weight updates are calculated when pipelining is used as compared to no pipelining. In particular, four weight updates are calculated in Worker 2 until time τ when pipelining is used while only two weight updates are calculated during the same time interval with no pipelining, Fig. 1. Thus, pipelining increases the speed of training.

One of the challenges of model-distributed learning with pipelining is that it requires the storage multiple versions of weights. According to Example 2, $\mathbf{w}_2^{(t)}$ is stored at Worker 2 until $\mathbf{w}_2^{(t+4)}$ is calculated as $\mathbf{w}_2^{(t)}$ is needed to calculate error vectors $\delta_2^{(k)}(t), \ldots, \delta_2^{(k+3)}(t)$. Meanwhile, $\mathbf{w}_2^{(t+1)}, \mathbf{w}_2^{(t+2)}, \mathbf{w}_2^{(t+3)}$ are also calculated at Worker 2. This means that Worker 2 stores four weights $\mathbf{w}_2^{(t)}, \ldots, \mathbf{w}_2^{(t+3)}$ until finishing the calculation of $\delta_2^{(k+3)}(t)$. The number of weight versions that needs to be stored in workers is proportional with the mini-batch size (it is four in Example 2). We design `mcPipe` to address this issue in the next section. We note that model-distributed learning has other challenges such as (i) single point of failure, (ii) model partitioning and assignment to workers. These issues are complementary to our work and partially addressed in [1], [4].

C. `mcPipe`: *Pipelining for Memory-Constrained Edge Computing Devices*

We design `mcPipe` to reduce the storage requirements of model distributed learning. The main idea behind `mcPipe` is that weight updates are calculated only when they are needed. In the forward propagation phase of model-distributed learning with pipelining, the most recent weight is used to calculate the activation vector. On the other hand, in the back-propagation phase, error vectors are calculated using the same weights that their corresponding activation vectors use. For example, $\delta_2^{(k+2)}$ is calculated using weight $\mathbf{w}_2^{(t)}$ in Example 2, although $\mathbf{w}_2^{(t+1)}$ and $\mathbf{w}_2^{(t+2)}$ are available, because the corresponding activation vector of $\delta_2^{(k+2)}$, which is $a_2^{(k+2)}(t)$, is calculated using $\mathbf{w}_2^{(t)}$. Such an approach has shown to have nice convergence properties [1].

To reduce the number of weights stored in workers, `mcPipe` calculates the weights when they are necessary. For example, $\mathbf{w}_2^{(t+1)}$ is not calculated immediately after receiving $\delta_2^{(k)}(t)$ in Example 2. Instead, error and activation vectors needed to calculate the weight $\mathbf{w}_2^{(t+1)}$ are stored, i.e., $\delta_2^{(k)}(t)$ and $g(a_1^{(k)}(t))^T$. These vectors are used to calculate $\mathbf{w}_2^{(t+1)}$ when needed, i.e., before calculating a new activation vector. A calculated weight matrix is deleted from the memory if (i) no other error vector needs to use that weight ($\mathbf{w}_2^{(t)}$ is deleted from Worker 2 after $\delta_2^{(k+3)}(t)$ is calculated), or (ii) there are earlier weights (if $\mathbf{w}_2^{(t)}$ is still needed by Worker 2, $\mathbf{w}_2^{(t+1)}$ is not stored). Thanks to storing only activation and error vectors instead of weight matrices, `mcPipe` reduces the memory cost significantly as formally stated in the next theorem.

Theorem 1: Assume that the total model size is Ω, which is shared equally across N workers. If the batch size is m, the memory cost of `mcPipe` at each worker is at most $\frac{\Omega}{N} + 2(m-1)\sqrt{\frac{\Omega}{N}}$, while it is $\frac{m\Omega}{N}$ for model distributed-learning via pipelining.

Proof: `mcPipe` stores one weight matrix and multiple activation and error vectors. The memory cost of a weight matrix is $\frac{\Omega}{N}$ as we assumed that the model is equally distributed across workers. The cost of storing a vector is $\sqrt{\frac{\Omega}{N}}$. In the worst case, we store $m-1$ activation and error vectors, so the total cost becomes $\frac{\Omega}{N} + 2(m-1)\sqrt{\frac{\Omega}{N}}$. Model-distributed learning via pipelining stores m weight matrices in the worst case scenario, so the cost becomes $\frac{m\Omega}{N}$. $\square$

As seen, mcPipe improves the memory cost on the order of m as compared to model-distributed learning via pipelining, which is significant for memory-constrained edge computing devices. We note that mcPipe introduces higher computing cost as activation and error vectors should be multiplied multiple times to provide on-demand weights. We show through real experiments in the next section that the impact of the additional computation is minimal on the convergence time.

V. Experimental Evaluation

We evaluate the performance of mcPipe on a real testbed and compare with different competing schemes. Our experimental environment consists of two MSI GS65 Stealth-483 computers and one 15" Macbook Pro (Core i7). The computers are connected to the same local area network using ethernet cables. We have used VGG16 [22] as our training model. VGG16 is a state-of-the-art deep neural network for image recognition.

In Fig. 2, we show the experimental results for a two worker scenario training the VGG16 model for the MNIST dataset. Both workers are MSI computers. The horizontal axis represents the training time in minutes, while the vertical axis represents the accuracy of the trained model. We show averages over 10 different runs of the same experiment with different weight initializations. We compare the performance of our scheme, mcPipe with different baselines and competing schemes. In particular, "Local" represents local computing where all training is performed on one of the MSI computers, which are both faster than the Macbook. "GreedyUpdates" refers to the scheme where the workers do not worry about the consistency about the gradient updates during the training phase. Specifically, when a new update opportunity arises at any worker at any given time, the worker updates its weights, erasing any previous versions of its weights from previous times.

We can observe that both our scheme mcPipe and PipeDream achieve almost exactly the same performance at all training times. This is expected as mcPipe guarantees almost the same performance with PipeDream with the extra benefit of reduced storage cost per worker. The minor accuracy difference stems from the extra multiplications of activation and error vectors to provide the on-demand weights. Moreover, both mcPipe and PipeDream outperform all competitors, especially at large training times. An interesting observation in this context is the optimality of GreedyUpdates at the initial phases (i.e. the first 80 minutes) of training. From this observation, we can conclude that approximately steering the weight vectors in the right direction becomes more important than achieving consistent weight updates during early stages of training. On the other hand, ignoring consistency becomes costly at the late stages of training where fine tuning of weights becomes important. In fact, we observe that GreedyUpdates appears to hit an error floor, while both mcPipe and PipeDream outperform GreedyUpdates by 5%, which is very significant at high accuracy levels. Local training performs poorly simply because it can only utilize the computing resources of a single machine, while data distributed training performs poorly as a result of the overhead of passing the model parameters across workers.

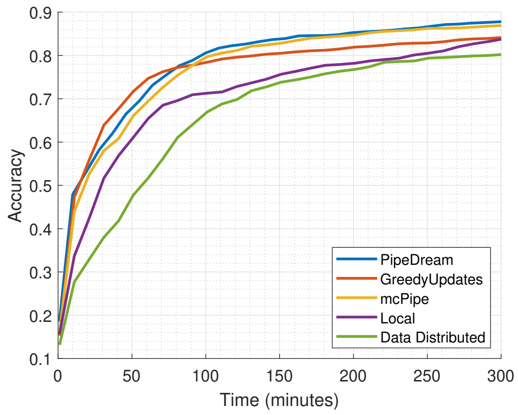


Fig. 2: Experimental results for a two-worker topology over the MNIST dataset.

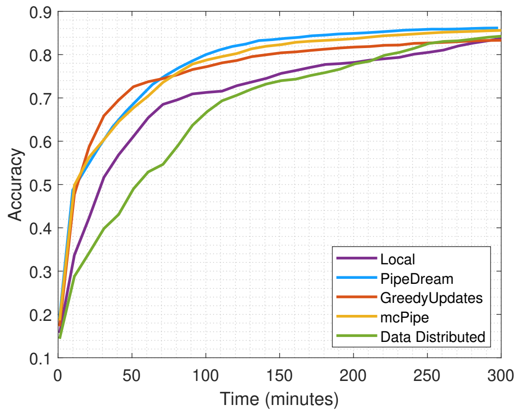


Fig. 3: Experimental results for a three-worker topology over the MNIST dataset.

Fig. 3 provides the results for the three-worker scenario. The general performance trend of difference schemes are similar to Fig. 2. For layer-distributed schemes such as mcPipe, accuracy at a given training time is slightly better, owing to the extra worker available at the system. For example, with mcPipe, reaching an

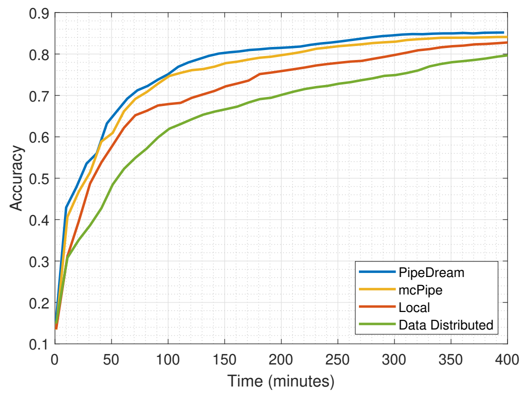


Fig. 4: Experimental results for a three-worker topology over the CIFAR-10 dataset.

accuracy of 60% takes 40 and 30 minutes for a 2 and 3 worker system, respectively.

In Fig. 4, we consider the same three-worker topology, but train the network over the more difficult CIFAR-10 dataset. We observe that the relative performance of different schemes are similar to the previous figures. On the other hand, as a result of training a more challenging dataset, the time required to reach a given level of accuracy is now longer, as compared to Fig. 3.

VI. Conclusion

We investigated a model-distributed DNN training for memory constrained edge computing devices. Existing model-distributed learning mechanisms store multiple versions of model weights for consistent gradient updates, which puts a strain on memory-constrained edge computing devices. We designed `mcPipe`, which uses on-demand weight updating policy to reduce the amount of stored weights. We implemented `mcPipe` in a real testbed and showed that it improves the memory cost without hurting convergence rate.

Acknowledgment

This work was supported in parts by the Army Research Lab (ARL) under Grant W911NF-1820181, National Science Foundation (NSF) under Grants CCF-1942878 and CCF-1814717, and the University of Illinois at Chicago Discovery Partners Institute Seed Funding Program.

References

[1] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv preprint arXiv:1806.03377*, 2018.

[2] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems*, 2019, pp. 103–112.

[3] C.-C. Chen, C.-L. Yang, and H.-Y. Cheng, "Efficient and robust parallel dnn training through model parallelism on multi-gpu platform," *arXiv preprint arXiv:1809.02839*, 2018.

[4] P. Li, E. Koyuncu, and H. Seferoglu, "Respipe: Resilient model-distributed dnn training at edge networks," in *IEEE ICASSP*, 2021.

[5] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.

[6] V. Smith, C.-K. Chiang, M. Sanjabi, and A. S. Talwalkar, "Federated multi-task learning," in *Advances in Neural Information Processing Systems*, 2017, pp. 4424–4434.

[7] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konecny, S. Mazzocchi, H. B. McMahan *et al.*, "Towards federated learning at scale: System design," *arXiv preprint arXiv:1902.01046*, 2019.

[8] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[9] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, "When edge meets learning: Adaptive control for resource-constrained distributed machine learning," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 63–71.

[10] ——, "Adaptive federated learning in resource constrained edge computing systems," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1205–1221, 2019.

[11] H. Li, K. Ota, and M. Dong, "Learning iot in edge: Deep learning for the internet of things with edge computing," *IEEE network*, vol. 32, no. 1, pp. 96–101, 2018.

[12] K. Portelli and C. Anagnostopoulos, "Leveraging edge computing through collaborative machine learning," in *2017 5th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*. IEEE, 2017, pp. 164–169.

[13] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," in *Advances in Neural Information Processing Systems*, 2017, pp. 1709–1720.

[14] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," in *Advances in neural information processing systems*, 2017, pp. 1509–1519.

[15] J. Langford, L. Li, and T. Zhang, "Sparse online learning via truncated gradient," *Journal of Machine Learning Research*, vol. 10, no. Mar, pp. 777–801, 2009.

[16] X. Yuan, P. Li, and T. Zhang, "Gradient hard thresholding pursuit for sparsity-constrained optimization," in *International Conference on Machine Learning*, 2014, pp. 127–135.

[17] J. Wangni, J. Wang, J. Liu, and T. Zhang, "Gradient sparsification for communication-efficient distributed optimization," in *Advances in Neural Information Processing Systems*, 2018, pp. 1299–1309.

[18] D. Alistarh, T. Hoefler, M. Johansson, N. Konstantinov, S. Khirirat, and C. Renggli, "The convergence of sparsified gradient methods," in *Advances in Neural Information Processing Systems*, 2018, pp. 5973–5983.

[19] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[20] A. Gormez and E. Koyuncu, "Class means as an early exit decision mechanism," *arXiv preprint arXiv:2103.01148*, 2021.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[22] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR), May 2015*, 2015.