Optimizing Distributed Training on Intermittently Connected Networks

Ronak Malik, Albert Qi, Steve Dalla

***Abstract*—As machine learning workloads and dataset sizes continue to grow, machine learning training will inevitably become more distributed. Whether this is through techniques such as highly distributed training on edge devices or just the expansion to multiple datacenters, proper optimization of these distributed systems will become increasingly critical. However, this optimization needs to be applicable for different scenarios, including ones with slow or unstable networks. It is not reasonable to assume that distributed training will only ever be run across good, stable connections, and as such, it is increasingly important that strategies for optimizing distributed training on intermittently connected networks are developed.**

There are a few strategies that we test. We first experiment with decreasing the frequency of weight aggregation of an all-reduce algorithm, seeing how it affects accuracy and performance on a simple triangle network. We also experiment with gossip learning, with both random neighbor communication and network-aware communication, and compare it to our previous strategies. Then, we test all of these strategies on more complex networks with a higher number of nodes. On a simple triangle network, performing all-reduce every four iterations results in the fastest time-to-accuracy, ultimately converging 53.8% faster than standard all-reduce and 43.7% faster than the best gossip learning strategy. On a network consisting of nine nodes, performing all-reduce every two iterations results in the fastest time-to-accuracy, ultimately converging 28.3% faster than standard all-reduce while gossip learning fails to converge.

I. Introduction

Highly distributed machine learning is becoming ever more relevant. From the scale of millions of edge devices training a single model or simply considering multiple, geographically separated datacenters that are connected through a network, the agents in these distributed systems are moving farther apart. While much of the current literature assumes we have strong and homogeneous networks, this assumption does not hold for many important situations. It is not always guaranteed that the network between geographically isolated datacenter locations will be strong. On the other extreme, federated learning devices could be connected via spotty, wireless links with highly heterogeneous networks between them.

This paper examines whether existing techniques to improve distributed machine learning workloads hold for these extreme cases of network intermittency and whether there exists better methods for improving training in these cases. For example, the standard method of pipeline parallelism will clearly suffer severe performance penalties from network failure or delay because it requires a synchronization step between every forward and backwards pass. However, through localization techniques, this problem could be avoided. Similarly, we explore other techniques such as gossip learning and varied weight aggregation rates.

Some of the main challenges in optimizing training over poor networks are 1) network unpredictability, 2) context-dependent tradeoffs between optimization and model accuracy, and 3) the possible issue of stragglers.

1) Since we are training over poor networks, the results that we get may be unpredictable. To handle this issue, we use a simulation environment that allows for determinism.
2) Increasing performance of training comes at the expense of decreasing accuracy. The optimal balance between the two depends on the situation and context.
3) If aggregations are not performed on every epoch, then some nodes may lag behind others and stray far away from the global model. This issue could be diminished via vector clocks, ensuring that no node is significantly behind.

Something that is as challenging as actually creating algorithms that are optimized for intermittent networks is modeling the networks themselves. Determining how a network should behave that is both unpredictable but also realistic is a significant system design decision.

Our system uses PeerSim, a simulation environment for peer-to-peer networks, in order to generate network topologies. PeerSim gives us an easily configurable environment to test our potential strategies. The simulation itself is done using custom a infrastructure that simulates random network latency and communicates with the training layer. The training is performed via PyTorch on the MNIST dataset.

Through this, we are able to test six different strategies on networks with both three and nine nodes. On a network with three nodes, the optimal strategy is to perform all-reduce every four iterations. This gives us the fastest time-to-accuracy (at 80.14 seconds), which is 53.8% lower than the time-to-accuracy for standard all-reduce (at 173.56 seconds). With nine nodes, the optimal strategy is to perform all-reduce every two iterations. This gives us the fastest time-to-accuracy (at 461.17 seconds), which is 28.3% lower than the time-to-accuracy for standard all-reduce (at 642.94 seconds).

Gossip learning does not perform as well as we initially expected, with random gossip learning converging in 142.41 seconds on a 3-node setup and both versions of gossip learning failing to converge on a 9-node setup. Part of this is likely due to the effect of stragglers that is exacerbated as the size of the network increases. Furthermore, gossip learning communicates over the network on every iteration, and there is some additional latency introduced as a result. These factors may lead gossip learning to perform worse than we originally anticipated.

Overall, we make the following contributions in this paper:

1) We highlight the need to optimize strategies for distributed machine learning training on intermittently connected networks.
2) We propose modified all-reduce algorithms and gossip learning algorithms in order to improve the time-to-accuracy of training.
3) We implement and evaluate these algorithms on a simple triangle topology and a larger network with nine nodes.

II. Design

We begin our strategy design by considering simple topologies such as a basic triangle setup. Through these insights, we then generalize our strategies to larger systems.

A. Simulation

The network is first generated by PeerSim, and nodes are initialized with a common, random model. We reduce the scope of the topologies to complete networks; we believe that we can do this without loss of generality because high latencies between nodes provide a similar effect to reducing the number of connections within a network. This allows us to simplify some of our algorithms such as all-reduce. It is paramount to the functionality of the system that all nodes start with the same model so that when models are combined, the only difference between them is the delta generated from training. Models are combined through a simple average since we weigh all of the training data equally and ignore data privacy.

Each node is assigned a unique section of the training data, which we call a shard. Each training iteration, which may or may not be synchronized with the iterations of the other nodes (depending on the strategy), a new section of the shard, or a mini-batch, is used to train the model. Training is performed using a standard CNN on the MNIST dataset using stochastic gradient descent. We use a stateless optimization algorithm because we do not store the optimizer state between iterations.

Every training iteration, we run the model against the MNIST provided test set. If any node in the network reaches the target accuracy, then the simulation is halted. We choose to stop the simulation based on the maximum accuracy of the network because the accurate model can then be distributed to the other nodes for continued training or inference. We initially considered halting the simulation when all nodes reached target accuracy, but we felt that 1) this would take unnecessarily long to run, and 2) it makes more sense to simply distribute the first model that reaches target accuracy.

To simulate network latency, we intercept communication between nodes and inject a random latency in the communication. The latency is generated by assigning an initial latency to each node uniformly between 1 and 5 seconds. Then, following each communication, the edge's latency is randomly shifted by a value between -0.5 and 0.5. We believe that this is a good estimation of a high latency network, as latency is highly variable and changes often but is still related to what the latency was recently. It should also be noted that network latency is not bi-directional, and there is a different latency between sending and receiving. This allows us to simulate a more realistic form of communication (for example, download and upload speeds may be different on the same network).

Each node runs its training/testing iterations and sharing protocol independently (and concurrently) from all other nodes, allowing for high fidelity testing of the different strategies. We opt for this approach instead of a cycle-driven simulation because the asynchrony and desynchronization of nodes is a significant factor in time-to-accuracy. Additionally, it is challenging to reason about how so-called "virtual" latencies should be applied to different strategies (for example, the overall latency of an all-reduce cycle is the maximum latency of that cycle, but this is not necessarily the case for other protocols).

Lastly, note that all of our randomness (both in PeerSim and PyTorch) is seeded in order to provide deterministic and replicable results. This helps mitigate the issue of unpredictability that comes with training on intermittently connected networks.

B. Evaluation Metrics

The primary evaluation metric we use is time-to-accuracy (TTA). In our testing, all nodes performed their training using the exact same hardware and the same mini-batch size, so the training and testing time for a single iteration is a constant factor in our evaluation (however, the number of iterations is still significant). Therefore, the most important metric to measure is how long it takes for the network to reach the target accuracy (in this case, 90%) because this includes the communication latency in between training cycles. Other metrics such as number of epochs do not take this into consideration. The injected latency by the simulation is a real-time delay, so the evaluation results presented in the next section are wall clock values.

C. Strategies

1) All-Reduce: The simplest strategy that we test is all-reduce, which aggregates models across all nodes every iteration. This keeps models up to date across nodes at the cost of very high communication overhead. While this is an optimal aggregation strategy in terms of model convergence, we expect the high network latency to significantly impact the performance, especially as the network size grows.

2) Decreasing Weight Aggregation Frequency: Another strategy that we consider is decreasing the frequency of weight aggregation in an all-reduce-like protocol. Performing global aggregations on every iteration is very costly and possibly unnecessary — combining models that are very far apart in training iterations may result in an overall worse model. Instead of aggregating every iteration, we aggregate every other iteration or every four iterations, for example. A delicate balance needs to be made in order to ensure that convergence is still possible in a reasonable time.

3) Gossip Learning: We also consider gossip learning, where instead of having nodes communicate with all other nodes, we can just have each node communicate randomly with its neighbors. We opt for this approach since the network between a node and its neighbor should typically be stronger than that between a node and another arbitrary node. On each iteration, if we communicate with a random half of a node's neighbors, for example, then the latency from communicating over the network should be smaller than the latency from performing something like all-reduce.

Furthermore, we consider an alternative variation of gossip learning where, to actually pick a random neighbor, we find the network strength between a node and its neighbors. These network strengths then represent the probability that we communicate with that neighbor; this strategy has the advantage of being network-aware. When the network is poor, we do not communicate as much, and when the network is strong, we communicate more frequently.

Suppose that we have an array of neighbors' latencies $A = [\ell_0, \ell_1, \ldots, \ell_{n-1}]$. Then, we want to eventually map each latency into a probability. We can start by finding the reciprocal of every latency ℓ_i, giving us:

$$B = \left[\frac{1}{\ell_0}, \frac{1}{\ell_1}, \ldots, \frac{1}{\ell_{n-1}}\right]$$

Note that we take the reciprocal because we want high latencies to correspond with low probabilities and low latencies to correspond with high probabilities.

Now, we can let T be the sum over the array B such that T equals the following:

$$T = \sum_{i=0}^{n-1} B[i] = \sum_{i=0}^{n-1} \frac{1}{\ell_i}$$

After calculating T, we can just divide every element in the array B by T in order to get a list of probabilities:

$$P = \left[\frac{1}{\ell_0 T}, \frac{1}{\ell_1 T}, \ldots, \frac{1}{\ell_{n-1} T}\right]$$

Using this formula, we can see that every neighbor i should have a $1/(\ell_i T)$ probability of being communicated with.

III. Evaluation

A. Experimental Setup

We run all of our tests on a 16-core, 32 thread AMD EPYC 7R32 with excess memory. Nodes run as threads within a single process for simplicity. All of the tests had fewer than 32 nodes, so threads did not interfere with each other during the simulation.

Our tests are all run with 30 mini-batches per shard. For the 3-node setup, we test model accuracy every iteration. For the 9-node setup, we test model accuracy every three iterations in order to speed up our evaluation. Lastly, note that we output timestamps alongside our iteration number in order to actually measure wall clock time.

Through this setup, we can run six different strategies. Four of these strategies are simply all-reduce performed at varied frequencies (specifically every 1, 2, 4, and 8 iterations). These are aptly named All-Reduce-1, All-Reduce-2, All-Reduce-4, and All-Reduce-8. The other two strategies are variations of gossip learning. The Gossip-Learning-Random strategy chooses its neighbors to communicate with completely randomly, while Gossip-Learning-Latency chooses its neighbors based on the network strength. We first run these six strategies on a network with 3 nodes, and afterwards increase the number of nodes to 9.

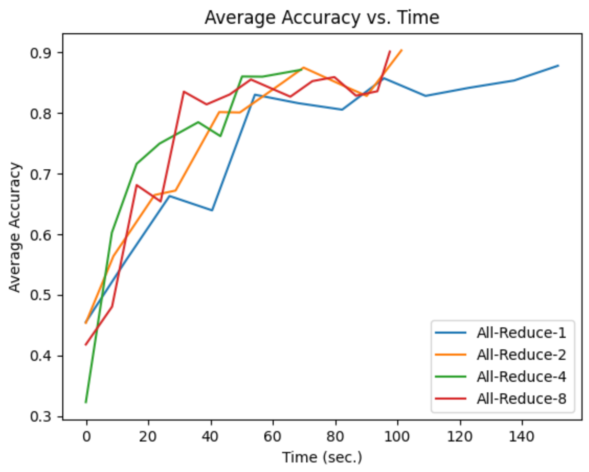

Fig. 1: Average accuracy over time for all-reduce with different aggregation frequencies on a network of 3 nodes.

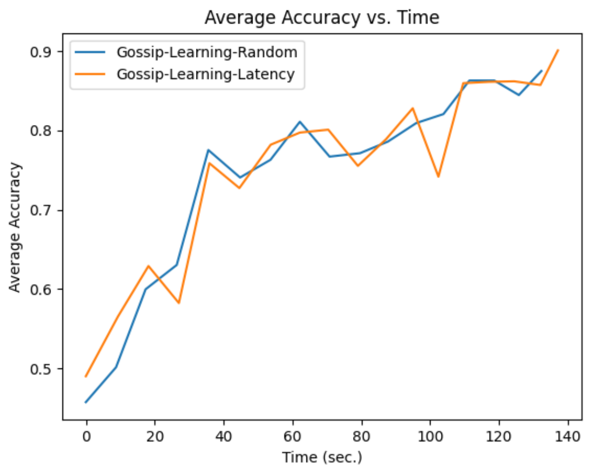

Fig. 2: Average accuracy over time for gossip learning on a network of 3 nodes.

TABLE I: TTAs on a 3-node setup.

Strategy	TTA (sec.)
All-Reduce-1	173.56
All-Reduce-2	110.92
All-Reduce-4	80.14
All-Reduce-8	106.69
Gossip-Learning-Random	142.41
Gossip-Learning-Latency	146.55

B. 3-Node Test Setup

We can first analyze the evaluation results of the four all-reduce strategies on the simple triangle topology. From Table I, we can see that standard all-reduce (i.e., All-Reduce-1) performs the worst with a time-to-accuracy of 172.56

seconds. All-Reduce-2 performs much better, with its TTA decreasing to 110.92 seconds and converging 35.7% faster than All-Reduce-1. Continuing on, we can see that All-Reduce-4 performs even better, reaching convergence in just 80.14 seconds and outperforming All-Reduce-1 by 53.8%. Lastly, All-Reduce-8 performs worse than All-Reduce-4 at a TTA of 106.69 seconds (on par with that from All-Reduce-2). Ultimately, we can conclude that the optimal strategy on a triangle setup is to perform all-reduce every four iterations.

These results are not too surprising. Standard all-reduce is expected to perform the worst as it suffers from the latency that arises from having to communicate over the network every iteration. When we decrease the all-reduce frequency to every two iterations, we observe better performance since we are communicating over the network less frequently. When we then decrease the frequency even further to every four iterations, our performance increases even more, as we are able to maintain an even better balance between local updates and all-reduces over the network. However, when we try to decrease the all-reduce frequency to every eight iterations, our performance worsens. In this case, while there is not that much latency from our calls over the network, the all-reduce frequency is so low that it takes more iterations to reach target accuracy. Thus, we want a happy medium between mitigating the effect of the network latency and still communicating enough between the nodes, and All-Reduce-4 provides the best balance out of the strategies.

Now, we can take a look at the gossip learning results in Table I. We see that Gossip-Learning-Random and Gossip-Learning-Latency perform fairly similarly with TTAs of 142.41 and 146.55 seconds, respectively. Looking at Gossip-Learning-Random (the better of the two strategies), we can conclude that it performs 17.9% better than All-Reduce-1 but 77.7% worse than All-Reduce-4.

This is rather surprising, as we initially expected the network-aware capabilities of Gossip-Learning-Latency to enhance its performance; instead, it performs worse than both All-Reduce-4 and Gossip-Learning-Random. There are two possible explanations for its suboptimal performance. First, gossip learning still communicates over the network on every iteration, meaning there is more latency here than in All-Reduce-4. Second, the asynchronous nature of gossip learning may cause some issues. Namely, because we prioritize communication with low latency nodes, nodes with high latencies might not be communicated with for an extended period of time. This results in stragglers and stale models! Not only does this reduce the average accuracy, but if any fresh model aggregates with a very stale one, then the accuracy of the fresh model may drastically decrease. Because of these reasons, Gossip-Learning-Latency seems to perform worse than initially expected.

C. 9-Node Test Setup

Let us now analyze the evaluation results of the all-reduce strategies with nine nodes. We can see from Table II that All-Reduce-1 performs the worst with a time-to-accuracy of 642.94 seconds. All-Reduce-2 performs much better with a

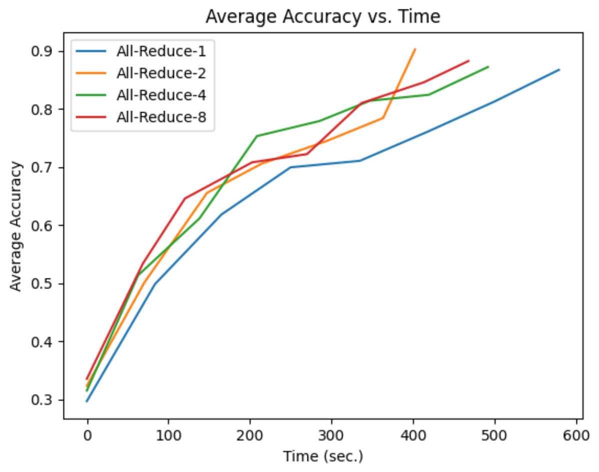

Fig. 3: Average accuracy over time for all-reduce with different aggregation frequencies on a network of 9 nodes.

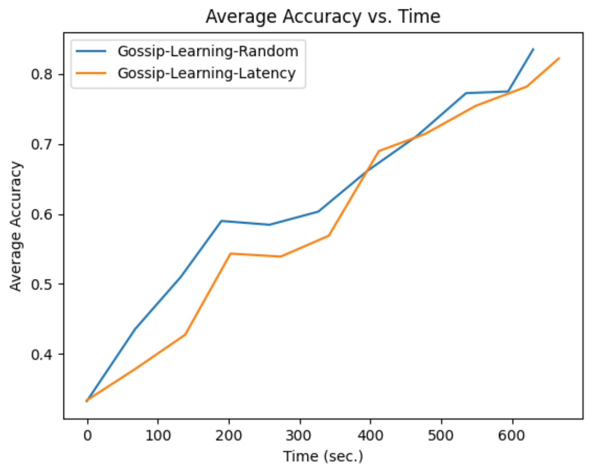

Fig. 4: Average accuracy over time for gossip learning on a network of 9 nodes.

TABLE II: TTAs on a 9-node setup.

Strategy	TTA (sec.)
All-Reduce-1	642.94
All-Reduce-2	461.17
All-Reduce-4	554.19
All-Reduce-8	531.95
Gossip-Learning-Random	N/A
Gossip-Learning-Latency	N/A

TTA of 461.17 seconds, converging 28.3% faster than All-Reduce-1. Afterwards, both All-Reduce-4 and All-Reduce-8 perform worse, taking 554.19 and 531.95 seconds to reach convergence, respectively. We can ultimately conclude that the optimal strategy with nine nodes is to perform all-reduce every two iterations.

These results are not too surprising, as they follow logic that is similar to that in the simple triangle setup. All-Reduce-1 performs the worst since there is a lot of latency that is introduced from communicating over the network every iteration. Furthermore, All-Reduce-4 and All-Reduce-8 do not perform as well as All-Reduce-2 since they communicate with other nodes too infrequently and thus require more iterations to reach convergence.

It is important to note that when we switch from three nodes to nine, the optimal strategy changes from All-Reduce-4 to All-Reduce-2. One possible explanation comes from the sharding of our data. When we have more nodes, each node will train on a smaller fraction of the data. As such, communication between nodes becomes more important. Thus, it makes sense that communicating more frequently, even with the added latency, is actually more beneficial.

Additionally, Table II tells us that neither Gossip-Learning-Random nor Gossip-Learning-Latency converges to our 90% accuracy threshold. However, we can also see from Figure 4 that, just like in our simple triangle setup, Gossip-Learning-Random consistently performs better than Gossip-Learning-Latency.

We believe that the poor performance of gossip learning is primarily due to the latency that is introduced in the network. As previously mentioned, gossip learning communicates over the network on every iteration, which in turn increases the latency experienced when compared to other strategies such as All-Reduce-2 or All-Reduce-4. We also run into the same issue of stragglers and stale models. However, because of the increased number of nodes, the issue of stragglers in gossip learning is only exacerbated. The quality and timeliness of information exchanged between nodes is crucial for convergence since we only communicate with a random subset of neighbors for each node in our algorithm. With more nodes, though, stragglers end up communicating with more neighbors. This negatively impacts the average accuracy and can cause our fresh models to worsen as well. Thus, the combined effect of network latency, the presence of stragglers, and the circulation of stale models all likely hinder our ability to achieve the desired 90% accuracy threshold through our gossip learning strategies.

IV. Related Work

A. Gaia

There already exist solutions for training across multiple datacenters, one of which is Gaia [1]. Gaia is a geo-distributed ML system that decouples communication within datacenters from communication between datacenters, building on the parameter server architecture. A main component of Gaia is dealing with the limited bandwidth provided by wide-area networks (WANs). It attempts to minimize communication between datacenters while at the same time trying to maintain the correctness of the ML algorithm. Gaia employs a synchronization model called Approximate Synchronous Parallel (ASP) to help do this.

The first component of ASP is the significance filter. This takes in a significance function and initial significance threshold from the programmer. These two work in conjunction to determine whether updates are significant, which is determined when the output from the significance function is larger than the threshold. ASP aggregates updates that are determined to be "insignificant" by the programmer until the aggregated updates reach significance. ASP then allows the parameter server to push the aggregated update.

ASP also utilizes a selective barrier to ensure that the WAN bandwidth does not bottleneck communication between datacenters. The selective barrier is used when the rate of a significant aggregated update is higher than the WAN bandwidth can support. When this happens, the parameter server initially sends only the indices of the update through the selective barrier. Which then blocks the receiver from reading the parameters until it receives the values of the update.

Finally, ASP uses a mirror clock. This helps ensure that worker machines are aware of significant time updates. Each datacenter's server reports its progress to corresponding servers in other datacenters. A check then sees if a server is too far ahead of the slowest server that shares the same parameters. If it is, it blocks work for its local machines until the slower servers catch up within some threshold.

It is important to note that Gaia is implemented over good, stable networks; what we are working with in our project is intermittent networks.

B. Gossip Learning

Gossip learning is a relatively nascent style for fully decentralized machine learning, and not many implementations are actually available. Our implementation is based off of the paper "Gossip Learning with Linear Models on Fully Distributed Data" [2], but nonetheless, the engineering decisions of our own gossip learning variations are completely determined by us.

The paper discusses gossip learning, a machine learning approach for peer-to-peer networks where data is fully distributed. In the paper's model, each network node holds a single data record, and raw data cannot be moved due to privacy concerns. The main challenge the authors face is learning without the ability to combine local models centrally. The problem is addressed through gossip learning by having multiple models perform random walks across the network, being updated with the local data at each node.

For the purposes of our paper, we specifically look at the gossip learning section, where we gain insight into the algorithm and how exactly it works. A key component of gossip learning is that at each node, the same algorithm is run. The algorithm consists of an active loop of periodic activity and a method for the handling of incoming models. Nodes handle these incoming models by creating a new model, combining it with the previously received model, and then storing the new model in a cache. The newest model in this cache becomes the model that is then sent to a randomly selected node. An important thing to note throughout all this is that synchrony between nodes is not assumed. This means that each of the nodes are running independently without coordination in timing of sending information.

The paper then focuses on two abstract methods within the gossip learning framework: "SELECTPEER" and "CREATEMODEL". SELECTPEER is integrated with a peer sampling service, allowing for the uniform and random selection of peers within the network. CREATEMODEL is used for updating and merging models based on locally available information.

V. Future Work

In the future, we would like to diminish the issue of stragglers by implementing vector clocks. Similar to Gaia's mirror clock, the introduction of vector clocks could ensure that no single node is significantly behind the others. If it is, then we could try to block training on the others nodes until the accuracy catches up to some threshold. This might improve some of the gossip learning strategies, and it would be interesting to see how they compare to the all-reduce strategies.

Furthermore, there are other strategies that we would like to test. One such strategy is aggregating only when models deviate by some factor of ϵ. This could be a modification to our gossip learning algorithm, for example, which currently communicates over the network on every iteration. This might help mitigate the effect of latency, similar to how the decreased frequency of all-reduce can improve the time-to-accuracy. Additionally, this strategy might be able to help address the issue of stragglers. By only sharing a model's weights on the significant changes to the model, stale updates might end up circulating less!

Additionally, we would like to run our tests on different network topologies. All of our current tests are run on fully connected networks, but it would be interesting to test our strategies on networks with fewer connections. This would allow for a more realistic network simulation, as it is not likely that all nodes will be connected to each other. By running tests on non-connected graphs, we gain additional insight into how efficiently our algorithms propagate data and whether a specific algorithm does so better than another.

VI. Conclusion

Machine learning training is becoming more and more distributed as machine learning workloads and dataset sizes grow increasingly large. A lot of the current literature assumes that we are working with strong and homogeneous networks, but this is not always guaranteed in many real-world scenarios. As a result, it is paramount that distributed machine learning training is properly optimized over a wide variety of situations, including those with intermittently connected networks (e.g., federated learning devices with highly heterogeneous networks between them).

We test a few different strategies, starting with the standard all-reduce algorithm on a simple triangle topology. We then experiment with performing all-reduce with a variety of different frequencies (namely every 2, 4, and 8 iterations), seeing how doing so impacts accuracy and performance. Afterwards, we also look at gossip learning based on a random subset of neighbors and a subset of neighbors based on the network strength, seeing how it compares to our aforementioned all-reduce algorithms. To then scale up our tests, we run these strategies on a bigger network with nine nodes instead of three. On a simple triangle network, doing all-reduce every four iterations leads to the fastest time-to-accuracy, ultimately converging 53.8% faster than standard all-reduce. With nine nodes, doing all-reduce every two iterations leads to the fastest time-to-accuracy, ultimately converging 28.3% faster than standard all-reduce.

References

[1] N. V. Kevin Hsieh, Aaron Harlap, *Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds*. USENIX Association, 2017.

[2] M. J. Robert Ormandi, Istvan Hegedus, *Gossip Learning with Linear Models on Fully Distributed Data*. IEEE, 2012.