EnsembleSync: Reliable Distributed SGD with Less Communication

Michael Xie, Michihiro Yasunaga

Abstract

Traditionally, distributed large scale training of neural networks splits data across workers. Every step of training requires communicating model parameters to and from a central parameter server, which may be very expensive for modern models. Our insight is that in machine learning, we can relax the conditions for consensus. Different parameters can be reconciled by ensembling rather than requiring a majority to agree on a value. Specifically, we consider EnsembleSync, a synchronous distributed training algorithm where each worker computes T steps of local training before sending updates to the parameter server, which then reconciles the updates by ensembling (averaging) the parameters. To avoid being bottlenecked by slow/faulty workers during synchronization, we dynamically reallocate more compute (GPUs) for a slower worker under a fixed budget. We ensure progress even with faults on up to $M-1$ workers by reallocating resources when workers fail. On top of computation and reliability gains, EnsembleSync *increases* the model performance with respect to base distributed training algorithms.

1 Introduction

Modern machine learning algorithms require an ever-increasing amount of training data and model parameters. For instance, ImageNet [2], a standard image recognition dataset, consists of 14 million web images (154 GB). ResNet [3], a widely-used neural network model for image classification, has 70 million parameters (244 MB). To scale to bigger datasets and models, recent works have explored distributing the training workload across multiple machines [8, 1, 7].

The typical distributed machine learning setup considers *data parallelism*, which parallelizes across the dataset (Figure 1) [1]. The training data is split into M shards and distributed across M *workers*. These workers compute model updates using their data shard and model. These updates are communicated to a central *parameter server*, which aggregates the updates. Before computing a new update, the workers pull the updated parameters from the server.

Distributed model training has two major challenges. First, existing training methods such as synchronous/asynchronous stochastic gradient descent (SGD) [8, 1] communicate model parameters with the server in every step of gradient descent so that all workers share the latest copies of parameters. However, as the model becomes more complex and requires more parameters, this communication becomes prohibitively expensive. Second, while synchronous algorithms linearize the gradient updates and are more principled, progress may stall when waiting for slow workers or faulty (dead) workers during synchronization.

We introduce EnsembleSync, a synchronous distributed SGD algorithm that reduces communication and is robust to slow and faulty workers. To reduce communication, EnsembleSync workers compute T steps of local training instead of a single step before sending updates to the parameter server. The parameter server reconciles the different parameters by ensembling, which computes a function (e.g. average) of the parameter versions rather than requiring a majority of the workers to agree on a single value (as in standard consensus algorithms

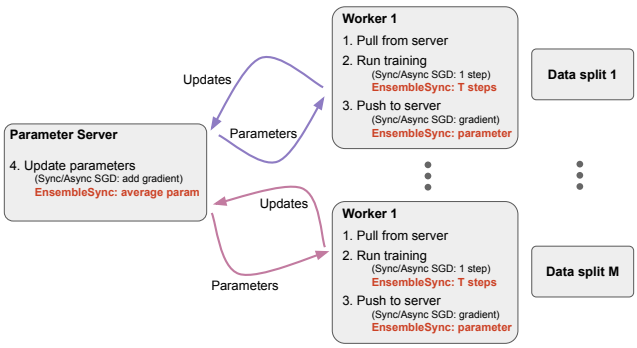

Figure 1: Distributed stochastic gradient descent (SGD) with data parallelism. Typical distributed SGD algorithms require communicating the model parameters between the parameter server and the workers in every training step, which is expensive for large models. EnsembleSync workers compute T steps of local training before sending updates to the parameter server, reducing total communication by a factor of T.

[5, 6]). This reduces the amount of communication to $1/T$ of the existing distributed training algorithms, while maintaining or even improving the accuracy of the trained model from regularization effects of ensembling. To ensure progress in the presence of slow or faulty workers, we dynamically reallocate more compute resources (GPUs) for slower workers under a fixed budget. To tolerate up to $M-1$ faulty workers, we redistribute the data splits (and compute resources) from dead workers to alive workers, sacrificing some parallelism for progress.

We evaluate the efficacy of our distributed training algorithm by using the ResNet-10 model [3] and the CIFAR10 image classification dataset [4]. In normal execution, EnsembleSync achieves better classification accuracy (84.29%) than synchronous and asynchronous SGD (83.19% and 82.96% respectively) while completing 100 epochs of training 6.6 times faster with 2 workers. With faulty workers, EnsembleSync continues to make progress while ensuring that the workers are synchronized, whereas workers in other algorithms either fall behind (asynchronous SGD) or must wait for dead workers to recover (synchronous SGD). With slow workers, EnsembleSync reallocates GPU resources to slow nodes according to their current computation speed, adjusting for differences in GPU speed and random network delays.

2 Preliminaries

In this section, we briefly describe distributed SGD algorithms. Comprehensive surveys of distributed machine learning are provided in [9]. In this work, we focus on the setting of centralized distributed training, where a central parameter server manages the de facto version of the model parameters, with data parallelism, which distributes computation across splits of a large dataset. The two most widely used algorithms in this setting are synchronous SGD (Sync SGD) and asynchronous SGD (Async SGD).

Distributed SGD algorithms. We describe the general framework shared by both Sync and Async SGD. Given a dataset $\{(x_i, y_i)\}_{i=1}^N$, a model f with parameters W, and a loss $L(y, f(x; W))$, the objective of model training is to find

$$W^\star = \operatorname*{argmin}_W \sum_{i=1}^N L(y_i, f(x_i; W)). \tag{1}$$

To optimize the parameters, stochastic gradient descent (SGD) iteratively updates the parameters W_t at timestep t via

$$W^{(t+1)} = W^{(t)} + \alpha \Delta W, \text{ where } \Delta W = -\nabla_{W^{(t)}} L(y_i, f(x_i; W^{(t)})) \tag{2}$$

and α is the learning rate. Figure 1 illustrates SGD in the distributed setting. Each worker $j \in \{1, \ldots, M\}$ first pulls the current parameter W from the parameter server (step 1), performs SGD locally on its own shard of data (step 2), and then push the parameter update ΔW_j to the server (step 3). Finally, the parameter server updates its parameter by adding ΔW_j (step 4).

Sync SGD. In synchronous SGD [1], the parameter server *waits* until every worker j finishes a step of SGD and reports to the server. Then the parameter server updates its parameters by summing all the ΔW_j,

$$W^{(t+1)} = W^{(t)} + \lambda \sum_{j=1}^{M} \Delta W_j, \tag{3}$$

where λ is our learning rate, and can be scaled to account for updates across multiple machines. We consider the beginning of training step i to start when a worker at training step i pulls the current parameters from the parameter server. The training step ends when all workers have reported their updates and the parameter server updates the parameters. In synchronous SGD, the beginning and end of each step are fully linearized.

Async SGD. In asynchronous SGD [8], on the other hand, the parameter server adds ΔW_j to the parameter *on the fly* as soon as receiving it from a worker j. This removes any locking and synchronization from the implementation, improving the computation speed. Thus, the beginning and end of each training step are not linearized, resulting in the possibility of a worker computing updates ΔW with respect to "stale" parameters from a previous step. This can cause a (small) drop in performance with the speed improvement [8].

3 Our Approach: EnsembleSync

3.1 EnsembleSync algorithm

We introduce EnsembleSync, a distributed SGD algorithm that only requires synchronization every T steps and adaptively reallocates resources to tolerate straggling or failed workers. The high-level steps (without resource reallocation) are as follows:

1. Each worker pulls the current parameters W from the server.
2. Each worker computes T steps of local SGD (instead of a single step).
3. Each worker j sends the resulting parameter W_j to the parameter server.
4. Server waits for all workers, then ensembles the updates W_j's, i.e., $W \leftarrow \frac{1}{M} \sum_{j=1}^{M} W_j$.

Figure 1 provides an illustration of this algorithm along with comparisons with Sync/Async SGD. Step 1 is blocking until all workers have reported from the previous sync round. This is implemented with a 2-phase locking procedure, where the parameter server manages a lock for each worker and releases the locks when all workers have reported for a round.

Comparison to traditional consensus. The workers operate as a distributed state machine across the synchronization points, where the state is the model parameters and epoch number. Each synchronization point requires the workers to agree on a single set of parameters. However, unlike traditional consensus algorithms [5, 6], we relax the problem by reconciling the parameters via ensembling. This forgoes the need to propose a consensus value, since each worker implicitly agrees to accept the outcome of the ensembling procedure. We only need 1 alive worker to report to compute the ensemble, resulting in robustness to $M-1$ worker failures.

3.2 Resource Reallocation

Workers may be slow due to different GPU and hardware types, or on a transient basis such as network connectivity and how busy the machine is. Since EnsembleSync is synchronized, slow workers can cause stalled progress at synchronization points. We consider using resource scheduling to dynamically reallocate compute resources at each sync point to adjust for slower workers. We focus on GPU memory reallocation, as the training speed of deep learning models depends heavily on GPU resources. Specifically, we consider the following implementation:

1. At the end of each training epoch, each worker reports their epoch execution time along with parameter updates.
2. Parameter server estimates the optimal GPU reallocation (described below).
3. Parameter server notifies the workers of the new reallocation configuration.

Suppose there are M worker nodes and a total GPU budget of B. We assume that the time between sync points for node i, t_i, has an inverse relationship with the amount of GPUs used on node i, x_i, such that $t_i = \alpha_i / x_i$, where α_i is a coefficient that may differ for every node.

For a particular sync point, assume that we have access to the true α_i for the next round of computation. In practice, we rely on estimates $\hat{\alpha}_i \approx \alpha_i$. We want to solve for the GPU allocations x_i such that the maximum computation time t_i is minimized,

$$\min_{x_1,\ldots,x_M} \max_i f_i(x_i) \quad \text{s.t. } \textstyle\sum_i x_i = B \tag{4}$$

where $f_i(x_i) = \alpha_i / x_i$ in our example. In general, f_i is a monotonically increasing or decreasing function that models the relation between computation and execution time. We show a lemma that allows for an analytical solution of this problem.

Lemma 1. *Suppose f_i is monotonically decreasing (increasing) for all i. Then the optimal solution $x_1^*,\ldots,x_M^*$ to* (4) *is such that for all i,j, $t_i^* = t_j^*$, where $t_i^* = f_i(x_i^*)$ and $\alpha_i > 0$ for all i.*

Proof. Let $x_1^*,\ldots,x_M^*$ be a solution where $f_i(x_i^*) = f_j(x_j^*)$ for all i,j, with objective value $\max_i f_i(x_i^*)$. We show that any other solution $x_1,\ldots,x_M$ must yield a larger objective value. Suppose for contradiction that $x_1,\ldots,x_M$ has a smaller objective value, such that $\max_i f_i(x_i) < \max_i f_i(x_i^*)$. Since f_i are monotonically decreasing (increasing), this implies that there is some j where $x_j > x_j^*$ ($x_j < x_j^*$). Since $\sum_i x_i = B$ for all solutions, this in turn implies that there is some k where $x_k < x_k^*$ ($x_k > x_k^*$). For this k, $f_k(x_k) > f_k(x_k^*)$ ($f_k(x_k) < f_k(x_k^*)$) by monotonicity, implying that $\max_i f_i(x_i) > \max_i f_i(x_i^*)$ and leading to a contradiction. □

In other words, the optimal computation time must be shared and equal between all workers. Specializing to $f(x_i) = \alpha_i / x_i$, this lemma leads directly to an analytical solution. Taking t^* to be the (shared and equal) computation time between all workers, the optimal solution must satisfy $x_i^* = \alpha_i / t^*$ for all i. Hence,

$$\textstyle\sum_i x_i^* = \sum_i \frac{\alpha_i}{t^*} = B \implies t^* = \frac{1}{B}\sum_i \alpha_i \implies x_i^* = B\frac{\alpha_i}{\sum_j \alpha_j}.$$

Intuitively, if we view α_i as the rate of slowdown for worker i, each worker is allocated a part of the GPU budget proportional to the fraction of the overall slowdown it contributes. We estimate α_i using the worker execution time in the last epoch and analytically compute the solution.

3.3 Fault Tolerance

Worker failures. To tolerate up to $M-1$ faulty (dead) workers, we let the parameter server redistribute the data splits from dead workers to alive workers. All operations update the `time_last_seen` for each worker. If for any worker, `time.now() - time_last_seen` is greater than `TIMEOUT`, server marks it as *dead*. This is implemented with a background thread that continuously checks `time_last_seen` for each worker. If the number of workers that have reported for the last sync point is equal to the number of currently *alive* workers, the background thread releases all locks to continue to the next round of training. If a dead worker communicates again, the server marks it as *recovered*. Recovered workers will receive the next epoch number and then fast forward to this epoch. It then blocks on retrieving updated parameters from the server until other workers reach the start of the epoch, at which point the worker is marked alive. At start of epochs, the server redistributes the data splits from dead workers evenly across alive workers, and runs GPU reallocation on alive and recovered workers to redistribute compute resources.

Parameter server failures. While not implemented in our system, it is possible to tolerate failure in the parameter server as well by checkpointing and replication. The server keeps checkpoints of model parameters at every sync point so it can reboot and resume, or if the server is replicated, a backup can take over using the checkpoint.

3.4 Natural Gradient Descent

Typically, model ensembles average the *outputs* of many models. Here, we average the parameters of the models, which may not correspond to one another. In order for parameter averaging to be an effective ensembling strategy, we constrain the networks to not diverge too far in output space via natural gradient descent, following Povey et al. [7].

	Sync SGD	Async SGD	EnsembleSync
Train time (min)	412.2	397.9	**60.95**
Test Acc. (%)	83.19	82.96	**84.29**

Figure 2: Training time and classification accuracies for the three algorithms during normal execution.

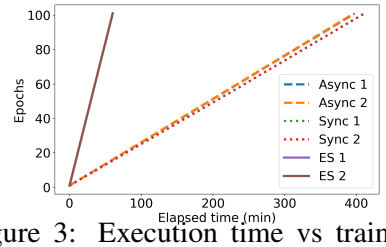

Figure 3: Execution time vs training epoch for the three algorithms.

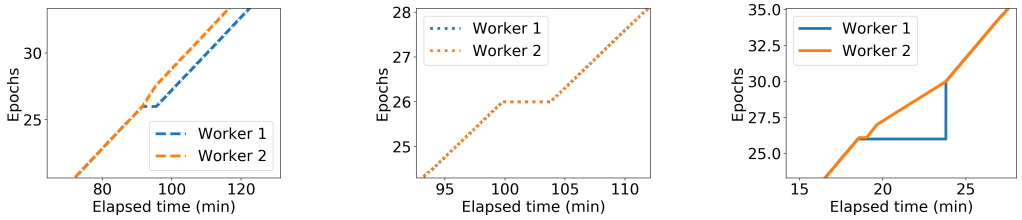

Figure 4: Execution time vs training epoch under a worker which fails and reboots. **(Left)** In Async SGD, the failed worker lags behind after rebooting. **(Middle)** In Sync SGD, other workers must wait for the failed worker to reboot. **(Right)** In EnsembleSync, the other workers continue after a small timeout, ensuring progress. The recovered worker gets the current epoch from the parameter server and skips forward.

4 Experiments

4.1 Experimental setup

We evaluate EnsembleSync on distributed training of a ResNet-10 model [3] on the CIFAR-10 image classification dataset [4] using PyTorch. In normal execution, EnsembleSync achieves better model performance than Sync or Async SGD while reducing the training time by $6.6\times$. We also demonstrate for slow and faulty workers that EnsembleSync uses resources reallocation to continue making efficient progress in these settings.

In all experiments, we use 2 workers that communicate with the parameter server via RPC primitives included in PyTorch. We take advantage of PyTorch's dynamic allocation of GPU memory according to GPU usage, allowing for fractional GPU allocation that changes during execution. For normal execution experiments, we use 2 workers with 1 GPU each (Tesla Xp and TITAN RTX). For faulty execution experiments, we use 2 workers both with GEFORCE GTX TITAN X GPUs. For all experiments, the two workers have 37500 data examples each from the CIFAR-10 training set (50000 examples), where they share 50% of the examples. We found that some overlap in the dataset splits is necessary in this smaller dataset to have enough samples to do local update steps well.

4.2 Normal Execution

Table 2 shows the execution time and classification accuracies on a test set for Sync SGD, Async SGD, and EnsembleSync. Async SGD finishes training slightly faster than Sync SGD due to removing synchronization, but still requires a long training time due to the communication cost. EnsembleSync finishes training about 6.6 times faster than Async SGD by reducing communication. In addition to making training faster, EnsembleSync improves classification accuracy by over 1% due to a regularization effect from ensembling the models.

4.3 Faulty Execution

We test the performance of the algorithms when the workers are slow or go down. In this experiment, we simulate failure of worker 1 at epoch 26 for 240 seconds and simulate 5 epochs of slow execution (increase each training step time by 0.5s) in worker 2 on epochs 5-15. We set `TIMEOUT` to 30 seconds and the total GPU resource budget to the GPU memory

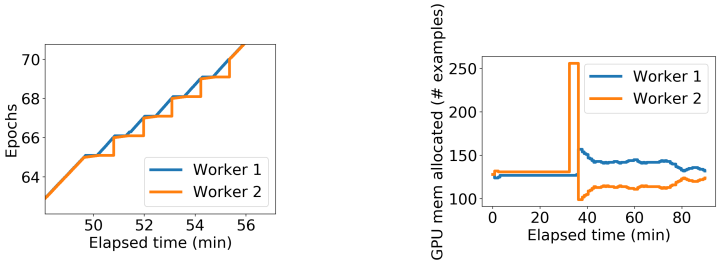

Figure 5: **(Left)** EnsembleSync maintains progress under repeated failures of a worker. **(Right)** GPU memory allocation (measured in number of parallel data examples held in memory) for EnsembleSync. At epoch 5, worker 2 becomes slow and more GPU resources are allocated for worker 2. At epoch 26, worker 1 fails and all GPU resources are allocated to worker 2 to continue the training.

needed to process 256 data examples at once. Each worker starts off with a batch size of 128 as an even distribution. The synchronization points are at 1/10 of the total epoch size.

Figure 4 shows a plot of execution time vs progress (in epochs) under a worker failure which triggers `TIMEOUT`. In Async SGD, the failed worker lags behind after recovering since each worker is independent. We found that the lagging worker finished 20 minutes later than the other. In Sync SGD, all workers must wait for the failed worker to come back online, pausing the training. In EmsembleSync, the worker failure triggers `TIMEOUT` on the parameter server, which releases all locks on other workers and allows them to continue making progress. Figure 4(d) shows that EnsembleSync maintains progress even under repeated failures of a worker.

Figure 5 plots the GPU memory allocation for the faulty execution run. At epoch 5, where worker 2 becomes slow, more GPU memory is allocated to worker 2. Later in epoch 26, when worker 1 fails, all the GPU resources are moved to worker 2. When worker 1 recovers, the resource allocation slowly approaches equilibrium.

5 Conclusion

We presented EnsembleSync, a synchronous distributed SGD algorithm that reduces communication via model ensembling and tolerates slow and faulty workers via resource reallocation. Our evaluation shows that EnsembleSync is 6.6 times faster and achieves better model performance than Sync SGD and Async SGD in normal execution, and it maintains steady progress in training in the presence of slow or faulty workers.

References

[1] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NeurIPS*, 2012.

[2] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[4] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2012.

[5] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 2001.

[6] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference*, 2014.

[7] Daniel Povey, Xiaohui Zhang, and Sanjeev Khudanpur. Parallel training of dnns with natural gradient and parameter averaging. *1410.7455*, 2014.

[8] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*. 2011.

[9] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. A survey on distributed machine learning. *ACM Computing Surveys*, 2020.