Distributed Machine Learning: A Comprehensive Review and Theoretical Approach to Enhancing Data Privacy & Security

Mark Lekina Rorat   Destin Niyomufasha

Dartmouth College

February 7, 2024

- **Distributed Machine Learning** involves training machine learning models across multiple computational resources by splitting the training process across machines, processors, or nodes, utilizing *data/model parallelism*, and employing coordination mechanisms such as *parameter servers* and *all-reduce* operations for effective synchronization.

Why do we need it?

- *Handling Large-Scale Data*: For datasets too large for a single machine.
- *Speeding Up Training*: Reduces training time through parallel processing.
- *Training More Complex Models*: Enables training of larger models not fitting in single GPU/CPU memory.
- *Resource Utilization*: Efficient use of computational resources across networks or clouds.
- *Geographical Distribution*: Trains on locally stored data, addressing privacy and bandwidth issues.
- *Fault tolerance*: Offers system resilience to node failures and scalable training capabilities.

Table of Contents

Background

- **Model vs. data parallelism**
- Centralized vs. decentralized optimization
- Synchronous vs. asynchronous scheduling
- Communication pattern used for exchanging parameters.

Model Parallelism

- Model parallelism involves partitioning a machine learning model into smaller parts or segments that can be processed in parallel.
- Simple implementation: encoder-decoder systems common in NLP, e.g., Transformer
- *Vertical partitioning*: applying splits between neural network layers.
- *Horizontal partitioning*: the layers themselves are partitioned.

Horizontal vs. Vertical partitioning

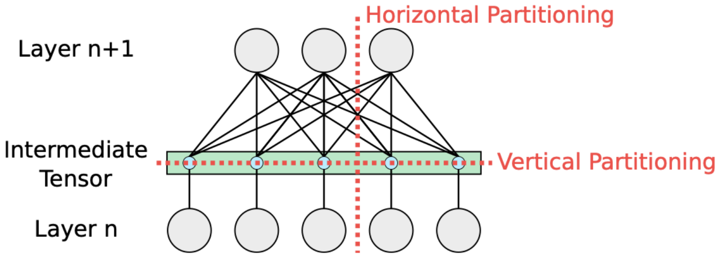


Figure 1: Horizontal vs. Vertical partitioning.[1]

[1]Source: https://arxiv.org/pdf/2007.03970.pdf

Challenges of Model Parallelism

- Requires meticulous synchronization of model segments to manage dependencies and ensure seamless data flow between partitions.
- High communication overhead, particularly for data-intensive exchanges between partitions.
- Scalability issues: as the number of partitions increases, the whole configuration needs to be restructured.

Data Parallelism

- Data parallelism involves splitting the dataset into smaller chunks and distributing these chunks across multiple processing units. Each unit then performs computations on its portion of the data, often using a copy of the same model.
- *Key idea*: the sum of per-parameter gradients computed using subsets of a mini-batch matches the per-parameter gradients for the entire input batch.

$$\frac{\partial L(x;w)}{\partial w} \approxeq \frac{\partial L(x^0;w)}{\partial w} + ... + \frac{\partial L(x^n;w)}{\partial w}$$

- Most recent distributed ML systems prefer data parallelism over model parallelism due to its ease of scalability.

Outline

- Model vs. data parallelism
- **Centralized vs. decentralized optimization**
- Synchronous vs. asynchronous scheduling
- Communication pattern used for exchanging parameters.

Centralized optimization

- Involves architectures where the optimization process (*i.e.*, the updating of model parameters based on computed gradients) is coordinated through a central node or a set of central nodes known as the parameter server(s).
- *Parameter-server*: optimization instance(s) that update model parameters and send new model to worker nodes.
- *Workers*: perform backpropagation and send computed gradients to the parameter server.
- This approach ensures that all worker nodes are synchronized with the same global view of the model parameters.
- Parameter servers, however, can be a bottleneck and single point of failure.

Centralized optimization

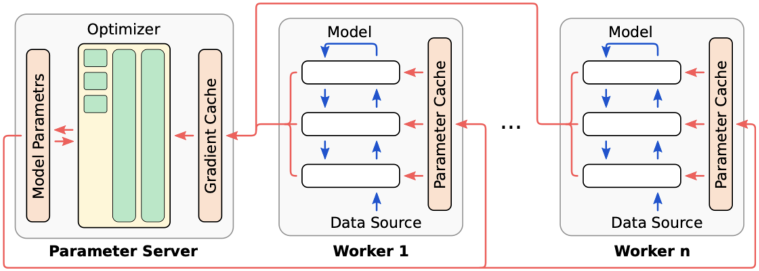


Figure 2: Workers evaluate the model to generate gradients (blue). The parameter server consumes them to update the model (red).[3]

[3]Source: https://arxiv.org/pdf/2007.03970.pdf

Decentralized optimization

- Refers to the approach where the optimization process is spread across multiple nodes without a central coordinator like a parameter server.
- Each node directly communicates with one or more other nodes in the network to exchange information (e.g., gradients, parameters) and update its local model.
- Significantly reduces bottlenecks associated with centralized approaches and improve fault tolerance, as there's no single point of failure.
- However, nodes may have different views of the model at a given time, which can lead to convergence issues.

Decentralized optimization

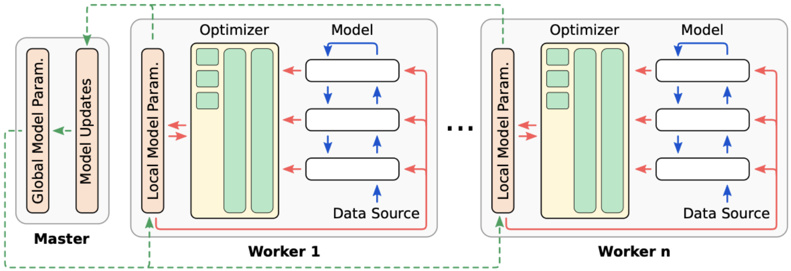


Figure 3: The master node generates the next global model by combining local model replicas (green) that were trained in isolation by the workers (red).[5]

[5]Source: https://arxiv.org/pdf/2007.03970.pdf

Exploration vs. exploitation phase

- Multiple independent workers concurrently try to solve a similar but not exactly the same problem.
- *Exploration phase*: workers iteratively evaluate the loss function using different mini-batches and independently update their local models.
- *Exploitation phase*: workers share their model updates with the master node, which merges the updates to distill adjustments that work better on average across the investigated portion of the training dataset, and shared the new parameters with the workers.

Exploration vs. exploitation phase

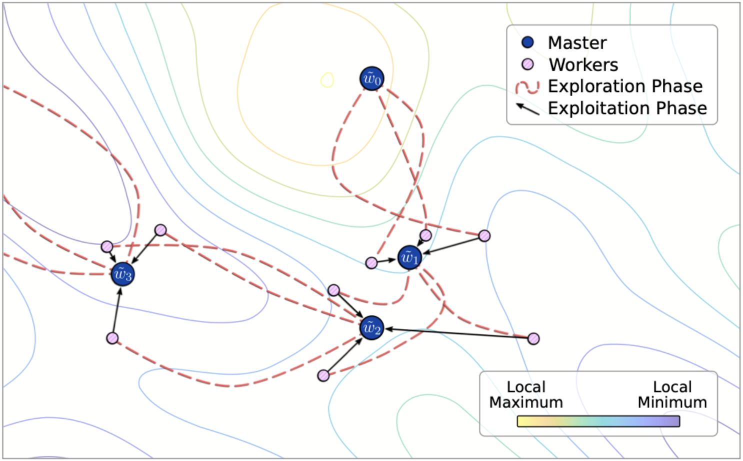


Figure 4: Workers start with the global model, then explore independently before combining their findings to refine the global model. [7]

Outline

- Model vs. data parallelism
- Centralized vs. decentralized optimization
- **Synchronous vs. asynchronous scheduling**
- Communication pattern used for exchanging parameters.

Synchronous Scheduling

- *Simultaneous updates*: All worker nodes must complete their tasks before the system proceeds to the next step to ensure that updates are based on the same model version.
- *Consistency and stability*: Ensures consistency in model updates, as all nodes synchronize at the end of each iteration, leading to potentially more stable convergence behaviors.
- *Potential for idle time*: Can lead to inefficiencies due to resource under-utilization if some nodes are slower than others, as faster nodes must wait for stragglers to complete before proceeding.

Asynchronous Scheduling

- *Independent updates*: Nodes update the shared model as soon as they complete their tasks, without waiting for other nodes. Updates are applied as soon as they are available, potentially speeding up the overall process.
- *Efficiency and scalability*: Maximizes resource utilization and efficiency by eliminating idle time, making it well-suited for environments with heterogeneous computing resources.
- *Risk of inconsistency*: Can lead to inconsistencies in the model state, as updates may be applied based on stale parameters, potentially causing slower convergence or divergence if not properly managed.

Decentralized Asynchronous Scheduling

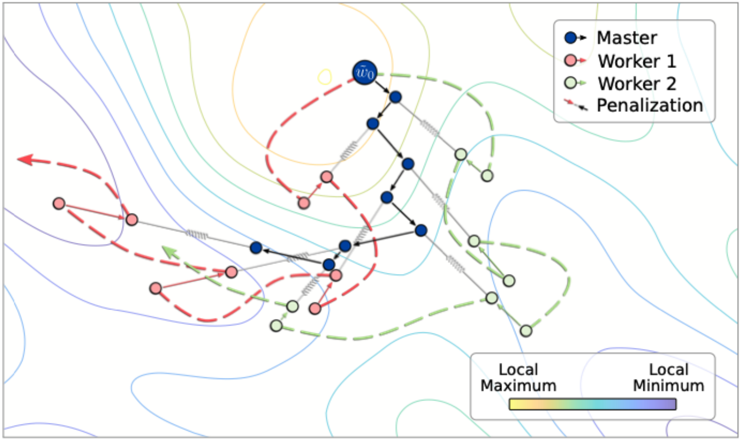


Figure 5: Workers explore locally, guided by a master node that steers them towards a shared optimum through a dance of attraction and penalties.[9]

[9]Source: https://arxiv.org/pdf/2007.03970.pdf

Bounded Asynchronous Scheduling

- *Delayed updates within bounds*: Allows for updates from nodes to be applied asynchronously, but within a certain bounded delay, ensuring that no update is based on information that is too outdated.
- *Balance between efficiency and consistency*: Aims to strike a balance between the efficiency of asynchronous methods and the consistency of synchronous methods, potentially reducing idle time while avoiding significant divergence in model states.
- *Complexity in implementation*: Implementing and maintaining a bounded delay can add complexity to the system, requiring mechanisms to track and manage the age of updates.

Outline

- Model vs. data parallelism
- Centralized vs. decentralized optimization
- Synchronous vs. asynchronous scheduling
- **Communication pattern used for exchanging parameters**

Communication patterns used for exchanging parameters

- Parameter Server
- All-Reduce
- Ring All-Reduce
- Gossip Protocols

Parameter Server

- As described earlier, one or more nodes hold global model parameters, whereas workers send gradients and receive updates from the parameter server.
- Scales to many workers and supports both synchronous and asynchronous updates.
- Potential bottleneck at parameter server.

All-Reduce

- Collective computation of an operation on data from all nodes. In the context of distributed ML, it aggregates gradients from all nodes to compute the global gradient.
- Ideal for synchronous data-parallel training, where gradients need global aggregation.

Ring All-Reduce

- A variant of the All-Reduce pattern where nodes form a logical ring and exchange data in a circular manner.
- Nodes in a ring exchange data with a neighbor, perform partial reductions and propagate the updates.
- Improves efficiency, since it reduces communication overhead and evenly distributes load to workers, making it ideal for large-scale training.

Ring All-Reduce

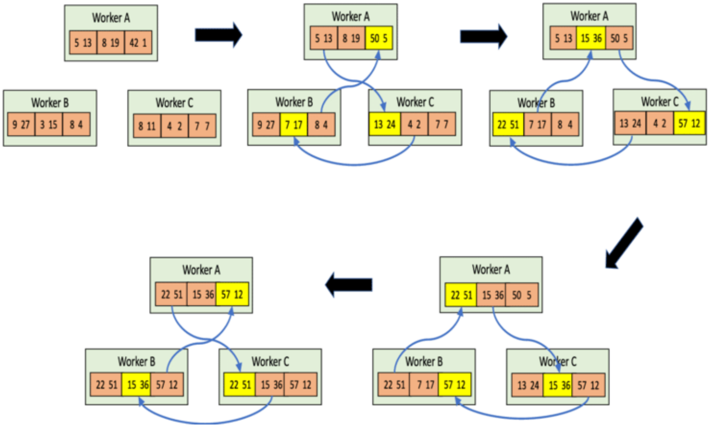


Figure 6: Nodes in a ring exchange gradients/parameters, performing on-the-fly reductions, until each has the final result.[11]

[11]Source: https://doi.org/10.1007/s42514-019-00018-4

Gossip Protocols

- Nodes randomly exchange parameters/gradients for decentralized convergence.
- Resilient to node failures, with good scalability.
- Ideal for decentralized training where strict order or central coordination is challenging.
- Convergence, however, may be slower or less predictable.

Security and Data Privacy in Distributed ML

- Proof of Learning
- Limitations of Proof of Learning

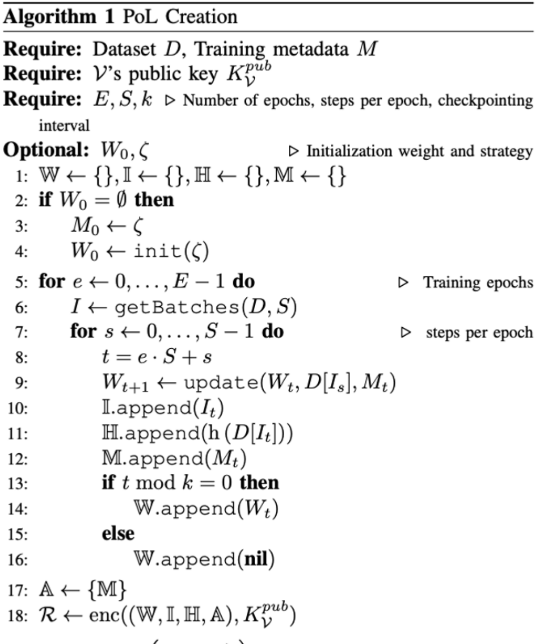
Proof of Learning

- In distributed ML, workers can be untrusted, and the model owner needs to verify that the workers have performed the required verifiable computations to obtain the model parameters.
- *Verifiable computation*: The proof should be verifiable by the model owner with high probability, ensuring that the worker has not cheated.
- *Key idea*: Verify the integrity of the training procedure used to obtain a machine-learning model as a means of ensuring adversarial workers do not poison the central model by sending corrupt gradient updates to the parameter server or other workers.

Proof of Learning

- With high probability, the parameter server can verify the proof and easily detect a dishonest worker.
- Verification process is computationally less expensive and model agnostic.
- An adversary cannot easily reconstruct secret information associated with the proof of learning since the computational cost should (ideally) be at least the same the cost of valid proof generation.

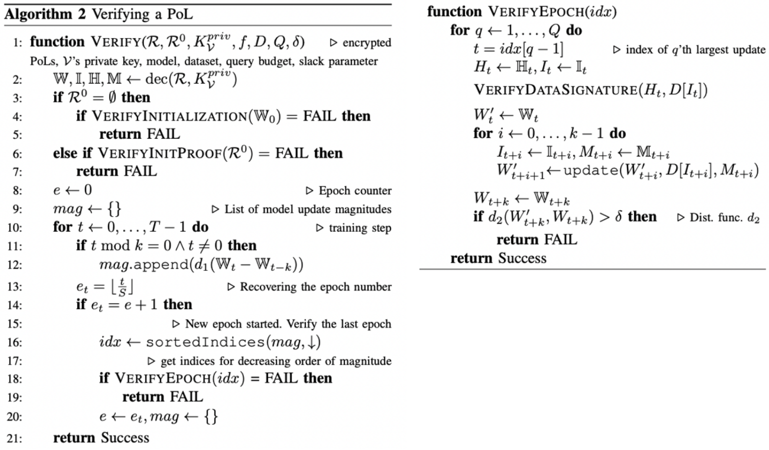
PoL Generation Algorithm



Algorithm 1 PoL Creation

```
Require: Dataset $D$, Training metadata $M$
Require: $\mathcal{V}$'s public key $K_{\mathcal{V}}^{pub}$
Require: $E, S, k$   ▷ Number of epochs, steps per epoch, checkpointing interval
Optional: $W_0, \zeta$   ▷ Initialization weight and strategy
 1: $\mathbb{W} \leftarrow \{\}, \mathbb{I} \leftarrow \{\}, \mathbb{H} \leftarrow \{\}, \mathbb{M} \leftarrow \{\}$
 2: if $W_0 = \emptyset$ then
 3:     $M_0 \leftarrow \zeta$
 4:     $W_0 \leftarrow \texttt{init}(\zeta)$
 5: for $e \leftarrow 0, \ldots, E-1$ do   ▷ Training epochs
 6:     $I \leftarrow \texttt{getBatches}(D, S)$
 7:     for $s \leftarrow 0, \ldots, S-1$ do   ▷ steps per epoch
 8:         $t = e \cdot S + s$
 9:         $W_{t+1} \leftarrow \texttt{update}(W_t, D[I_s], M_t)$
10:         $\mathbb{I}.\texttt{append}(I_t)$
11:         $\mathbb{H}.\texttt{append}(\mathrm{h}(D[I_t]))$
12:         $\mathbb{M}.\texttt{append}(M_t)$
13:         if $t \bmod k = 0$ then
14:             $\mathbb{W}.\texttt{append}(W_t)$
15:         else
16:             $\mathbb{W}.\texttt{append}(\textbf{nil})$
17: $\mathbb{A} \leftarrow \{\mathbb{M}\}$
18: $\mathcal{R} \leftarrow \mathrm{enc}((\mathbb{W}, \mathbb{I}, \mathbb{H}, \mathbb{A}), K_{\mathcal{V}}^{pub})$
```

PoL Verification Algorithm



Algorithm 2 Verifying a PoL

```
function VERIFY($\mathcal{R}, \mathcal{R}^0, K_{\mathcal{V}}^{priv}, f, D, Q, \delta$)    ▷ encrypted PoLs, $\mathcal{V}$'s private key, model, dataset, query budget, slack parameter
    $\mathbb{W}, \mathbb{I}, \mathbb{H}, \mathbb{M} \leftarrow \text{dec}(\mathcal{R}, K_{\mathcal{V}}^{priv})$
    if $\mathcal{R}^0 = \emptyset$ then
        if VERIFYINITIALIZATION($\mathbb{W}_0$) = FAIL then
            return FAIL
    else if VERIFYINITPROOF($\mathcal{R}^0$) = FAIL then
        return FAIL
    $e \leftarrow 0$    ▷ Epoch counter
    $mag \leftarrow \{\}$    ▷ List of model update magnitudes
    for $t \leftarrow 0, \ldots, T-1$ do    ▷ training step
        if $t \bmod k = 0 \land t \neq 0$ then
            $mag.\texttt{append}(d_1(\mathbb{W}_t - \mathbb{W}_{t-k}))$
        $e_t = \lfloor \frac{t}{S} \rfloor$    ▷ Recovering the epoch number
        if $e_t = e + 1$ then
                ▷ New epoch started. Verify the last epoch
            $idx \leftarrow \texttt{sortedIndices}(mag, \downarrow)$
                ▷ get indices for decreasing order of magnitude
            if VERIFYEPOCH($idx$) = FAIL then
                return FAIL
            $e \leftarrow e_t, mag \leftarrow \{\}$
    return Success
```

```
function VERIFYEPOCH($idx$)
    for $q \leftarrow 1, \ldots, Q$ do
        $t = idx[q-1]$    ▷ index of $q$'th largest update
        $H_t \leftarrow \mathbb{H}_t, I_t \leftarrow \mathbb{I}_t$
        VERIFYDATASIGNATURE($H_t, D[I_t]$)
        $W'_t \leftarrow \mathbb{W}_t$
        for $i \leftarrow 0, \ldots, k-1$ do
            $I_{t+i} \leftarrow \mathbb{I}_{t+i}, M_{t+i} \leftarrow \mathbb{M}_{t+i}$
            $W'_{t+i+1} \leftarrow \texttt{update}(W'_{t+i}, D[I_{t+i}], M_{t+i})$
        $W_{t+k} \leftarrow \mathbb{W}_{t+k}$
        if $d_2(W'_{t+k}, W_{t+k}) > \delta$ then    ▷ Dist. func. $d_2$
            return FAIL
    return Success
```

Figure 8: PoL Verification [13]

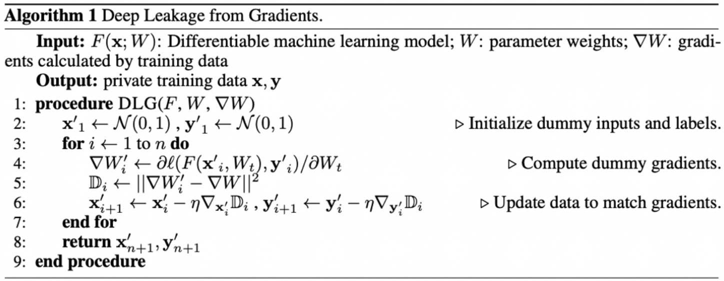
[13] Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9519402

Limitations of Proof of Learning

- Should we blindly trust the verifier (parameter server)?
- Does the *gradient sharing* scheme protect the privacy of the training datasets of each participant?
- The *gradient sharing* scheme is not secure against a malicious verifier (parameter server) who can reconstruct the private data from the gradients.
- It is possible to obtain the private data used in training from the provided gradients (Deep Leakage, Xu, et al.[14])

[14] https://proceedings.neurips.cc/paper/2019/file/60a6c4002cc7b29142def8871531281a-Paper.pdf

Deep Leakage Algorithm



Algorithm 1 Deep Leakage from Gradients.

Input: $F(\mathbf{x}; W)$: Differentiable machine learning model; W: parameter weights; ∇W: gradients calculated by training data

Output: private training data $\mathbf{x}, \mathbf{y}$

```
1: procedure DLG($F$, $W$, $\nabla W$)
2:     $\mathbf{x}'_1 \leftarrow \mathcal{N}(0,1)$ , $\mathbf{y}'_1 \leftarrow \mathcal{N}(0,1)$        ▷ Initialize dummy inputs and labels.
3:     for $i \leftarrow 1$ to $n$ do
4:         $\nabla W'_i \leftarrow \partial \ell(F(\mathbf{x}'_i, W_t), \mathbf{y}'_i)/\partial W_t$        ▷ Compute dummy gradients.
5:         $\mathbb{D}_i \leftarrow ||\nabla W'_i - \nabla W||^2$
6:         $\mathbf{x}'_{i+1} \leftarrow \mathbf{x}'_i - \eta \nabla_{\mathbf{x}'_i} \mathbb{D}_i$ , $\mathbf{y}'_{i+1} \leftarrow \mathbf{y}'_i - \eta \nabla_{\mathbf{y}'_i} \mathbb{D}_i$        ▷ Update data to match gradients.
7:     end for
8:     return $\mathbf{x}'_{n+1}, \mathbf{y}'_{n+1}$
9: end procedure
```

Figure 9: Approximating Training features from Gradients [15]

[15]Source: https://proceedings.neurips.cc/paper/2019/file/60a6c4002cc7b29142def8871531281a-Paper.pdf