An Overview of Computational and Communication Mechanisms for Scalable AI Systems

Sahil Tyagi

Table of Contents

Acronyms

AI Artificial Intelligence
API Application Programming Interface
ASIC Application Specific Integrated Circuit
CF Compression Factor
CNN Convolutional Neural Networks
CPU Central Processing Unit
DL Deep Learning
DNN Deep Neural Networks
DPO Direct Preference Optimization
FL Federated Learning
GD Gradient Descent
GPT Generative Pre-Trained Transformer
GPU Graphic Processing Unit
HPC High Performance Computing
IID Independent and Identically Distributed
IoT Internet-of-Things
LLM Large Language Model
MAC Multiplication and Accumulation
ML Machine Learning
MMU Matrix Multiply Unit
MPI Message Parsing Interface
NCCL NVIDIA Collective Communication Library
PS Parameter Server
RDMA Remote Direct Memory Access
RLHF Reinforcement Learning with Human Feedback
SGD Stochastic Gradient Descent
TCP Transmission Control Protocol
TPU Tensor Processing Unit

---

1. Introduction

With the advent of big data and Internet-of-Things (IoT), the number of smart devices has grown exponentially over the years. These devices capture data across a wide range of modalities, such as image/video in smartphones and surveillance camera feeds [298], audio and speech from smart speakers, text/language on phone/tablet keyboards, and more. This data can be applied to Artificial Intelligence (AI) in order to extract keys insights and make informed decisions. Machine Learning (ML) algorithms learn from given data to make decisions or predictions on unknown data, generally falling under one of the following tasks: *Regression*, *Clustering*, *Classification*, *Dimensionality Reduction* and *Anomaly Detection* [284]. Regression is used to relate a dependent variable to one or more independent variables. Clustering involves grouping data points by measuring proximity with respect to a specific field or property. Classification is done by categorizing data into one or more labels, while Anomaly detection helps detect outliers [215].

ML algorithms can be classified as one of the following types based on the type of feedback used for learning: *Supervised*, *Unsupervised*, *Semi-supervised* or *Reinforcement learning*. In supervised learning, a model learns a function that maps input data to a certain output. Each input datum has an associated output label and the learning function is tweaked accordingly for best fitting. In this way, the learned function can be used to predict output for new input data. A considerable challenge to resolve in supervised learning is the *bias-variance* trade-off. Bias comes from assumptions made by the learning technique to fit the target function. For e.g., a linear model will perform poorly if the underlying data has non-linear properties. On the other hand, variance comes from running the ML algorithm on a specific training set. Under high-variance, the performance of a learning technique varies with

the training subset. This is because the algorithm models the specific attributes of training samples itself rather than the underlying relationship between inputs and outputs. Regression and classification described earlier fall under supervised learning. Unsupervised learning aims to learn a function when there are no output labels associated with each input. The objective here is to capture the underlying structure of the data and group them together based on certain attributes. Applications of unsupervised learning include clustering, dimensionality reduction and generative models. Semi-supervised learning uses a mix of labeled and unlabeled data. Reinforcement learning trains an agent operating in an environment and take actions such that positive actions are rewarded and negative actions are penalized. This has applications in gaming, resource allocation and management, robotics, etc.

Of all the classes of ML algorithms, Deep Learning (DL) has garnered limelight by achieving state-of-the-art results in recent years. *Gradient Descent (GD)* [235] is the workhorse of DL models that faciliates iterative learning given the current loss and state of model parameters. Neural networks typically converge after multiple traversals over the entire training data, where each pass is referred to as an *epoch*. Convergence is heavily influenced by DL-specific variables or *hyperparameters* like batch-size, epochs, learning rate or step-size, activation function, weight decay, dropout, regularization, type of GD optimization used, etc. GD minimizes a loss function that compares ground truth with a model's predictions by adjusting model parameters in the negative direction of the gradient. A model with parameters w_t at iteration t, loss function $\mathcal{L}(\cdot)$ and learning rate η is updated at iteration $(t+1)$ as per Equation (1):

$$w_{t+1} = w_t - \eta \cdot \nabla f_t \quad \textit{where} \quad \nabla f_t = \frac{\partial \mathcal{L}}{\partial w_t} \tag{1}$$

In *Full GD*, gradients are computed over the entire training dataset at every iteration before GD update. Although the gradients are most accurate in this case, it is impractical as the per-iteration time is too high. As an alternative, *Stochastic Gradient Descent (SGD)* picks a random sample from the training subset to compute loss, gradients and perform SGD update. Additionally, SGD does not leverage vectorization of multiple samples on fast accelerators like Graphic Processing Unit (GPU)s. The gradients here are highly noisy as its taken from a single sample and thus, not representative of the true gradients computed over the entire training data. For a convex problem training to T iterations, SGD has been shown to have a convergence rate of $\mathcal{O}(1/T)$, while a non-convex problem like a neural network converges at $\mathcal{O}(1/\sqrt{T})$ [58], [181], [203]. *Mini-batch GD* is the optimal middle ground between the former two such that a batch is randomly sampled at every iteration for training. The steps involved in training of SGD-based algorithms are described as follows:

1.) Independent and Identically Distributed (IID) sampling of a batch of training data at every iteration.
2.) compute the loss between model's prediction and the ground truth.
3.) compute gradients, i.e., partial derivative of the loss function with respect to model weights or parameters.
4.) Post gradient calculation over backpropagation, apply updates to model parameters.

A. Prior Art in Deep Learning

Deep Neural Networks (DNN)s learn non-linear functions and relationships by repeatedly training on given data in a supervised/semi-supervised manner and learning via feedback. Although the first perceptron [232] was proposed in 1958, and Convolutional Neural Networks (CNN) [163] as well as long short-term memory (LSTM) [133] in the late 90s, some of the major breakthroughs have only happened at the turn of the last decade. Since then, CNNs dominate computer vision and segmentation benchmarks, while recurrent neural networks (RNN) like LSTM or attention-based transformers [283] like BERT [95] represent state-of-the-art approaches for time series prediction, machine translation, speech recognition and natural language processing. Back in 2012, AlexNet [158] won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [94] by achieving 83.6% Top-5 test accuracy. The model consists of 8 layers: 5 convolutional and 3 fully-connected (FC) layers, and used a novel activation function to introduce non-linearity at the time, called *rectified linear unit (ReLU)* [200]. Another key contribution of this work was to use general purpose graphic processing units (GPGPU) to accelerate training. AlexNet was trained on NVIDIA GTX 580 GPU with 3GB video memory. Following AlexNet, VGG16 [254] won the ILSVRC challenge in 2014, achieving 92.6% Top-5 test accuracy. The model comprises of 13 convolutional and 3 FC layers. The residual network called ResNet [131], introduced in 2015 used *skip connections* to solve the vanishing/exploding gradient problem that plagued contemporary deep networks. ResNet152 obtained 96.4% Top-5 test accuracy in the

2015 ILSVRC competition. Transformers [283] introduced the concept of attention in 2017 and laid the foundation for modern Large Language Model (LLM)s like OpenAI's Generative Pre-Trained Transformer (GPT)-4 [39], Google's BARD [34] and Meta's LLaMA [35]. The attention function of a transformer maps a query to a set of key-value pairs and produces an output. From [283], a transformer layer consists of a self-attention block followed by a two-layer fully connected network. The input consists of queries (Q) and keys (K) of dimension d_k. The output is calculated as a weighted sum of the values (V), where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key: Attention(Q,K,V) $= \texttt{softmax}(\frac{QK^T}{\sqrt{d_k}})V$. The original transformer model was shown to achieve a BiLingual Evaluation Understudy (BLEU) score [208] of 24.8 on WMT-14 (Workshop on Machine Translation dataset) English-to-German translation task and 41.8 on English-to-French translation.

B. Need for Scalable Distributed Training

According to OpenAI, the size of DL models is growing at such a pace that the computational requirements doubles every 3.5 months [32], while updated surveys estimate around 5-6 months [243]. The publication year and size of some leading DL models is described below:

- *2018*: GPT [217] with 100M and BERT [95] with 340M parameters.
- *2019*: GPT-2 [218] with 1B, XLNet [306] with 110M and Transformer-XL [89] with 257M parameters.
- *2020*: BART [166] with 140M, DialogGPT [319] with 1.5B, T5 [220] with 11B and Turing-NLG [33] with 17B parameters.
- *2021*: ViT (Vision Transformer) [99] with 86M-22B, GPT-Neo [68] with 20B and DALLE [223] with 12B parameters.
- *2022*: Stable Diffusion [231] with 890M, DALL-E2 [222] with 3.5B, BLOOM [300] with 176B, Megatron-Turing NLG [255] with 530B, Google's PaLM [229] with 540B and GLaM [102] with 1.2T parameters.
- *2023*: Sparrow [116] with 70B, GPT-3.5 [38] has 3 variants with 1.3B, 6B and 175B parameters, ChatGPT [36] with 175B, Google's PaLM-2 [267] with 340B.
- *2024*: GPT-4/4o [207] with 1.76T parameters, Anthropic's Claude-2 and 3 [22] with 20B, 70B and 2T variants, Google's Gemini [266] with 1.5T, Meta's LLaMA-2 and 3 [268] with variants between 2-70B parameters, and Claude Sonnet 3.5 [1].
- *2025*: OpenAI o1/o3 [16], DeepSeek-v3 [104] and DeepSeek-R1 with 671B total parameters [105].

From the trend above, it is evident that models have exponentially grown in size and complexity over time, and will continue to do so in the foreseeable future. Assuming each model parameter is stored as a single-precision (i.e., 32-bit) float, a model with 1 billion parameters would require over 4GB memory just to store model parameters. Scaling to a 1 trillion parameter model increases memory requirements to over 4000 GB. In addition, memory is also needed to hold the computation graph to store gradients and activation maps computed in backpropagation. Techniques like gradient checkpointing [120] trades memory for computation by flushing intermediate data from the computation graph to reduce memory utilization. This comes at the cost of increased computation as flushed activation maps need to be recomputed whenever needed. The enormous volumes of multimodal data is parsed and fed to these models one batch at a time. Although using large batch sizes is efficient and accelerates training, it further increases the memory consumption for DL training. Even if one could fit a model on a single node/device, the time taken to train such models on massive training sets is very high, sometimes taking days, weeks or even months [37]. It is thus crucial to develop distributed and parallelizable algorithms for training massive datasets on large and complex DNNs.

2. Integrating Deep Learning with High Performance and Distributed Computing

A. Hardware-Software Co-Design for DL Applications

1) Hardware developments

Training neural networks is compute-intensive, involving numerous Multiplication and Accumulation (MAC) operations on vectors, matrices or tensors in the forward pass, and gradient computation in the backpropagation phase. In the early days of deep learning, training was carried out on general purpose processors, i.e., Central Processing Unit (CPU). CPUs are designed to have a small number of general cores, while GPUs come with a large number of specialized cores. Additionally, CPUs are capable of executing a wide variety of instructions, while GPUs are designed to deal with data streams and work with a smaller set of instructions. GPUs were first built on a Single-Instruction-Multiple-Data (SIMD) model such that all cores could only execute the same piece of user program. Thus, they were initially not optimal for training neural networks. Eventually general purpose GPUs (GPGPUs) were built with a more flexible architecture and designed keeping parallel computing and High Performance Computing (HPC) in mind. Libraries like cuBLAS [12] were developed to bring GPU-accelerated implementation of basic linear algebra subroutines (BLAS) for AI and ML applications. The numerous MAC operations performed during training can thus be accelerated over the many cores of general purpose GPUs. Thus, adding GPUs for training can essentially be viewed as *scaling up* the training infrastructure. The GPGPU NVIDIA GTX 580 was first used for training AlexNet in the 2012 ImageNet challenge, and turned out to be significantly faster than training on CPUs [75].

There has also been considerable development in Application Specific Integrated Circuit (ASIC) for accelerating neural network training. Compared to CPUs and GPUs, such ASICs have more optimal performance, better energy efficiency, or both. One popular ASIC developed for deep learning is Google's Tensor Processing Unit (TPU) [206]. A Matrix Multiply Unit (MMU) based on a systolic array lies at the heart of the TPU. The ASIC uses a Multiple-Instructions-Multiple-Data (MIMD) architecture which allows for execution of several instructions over multiple data streams. TPU v1.0 was shown to be 15-30$\times$ faster than a NVIDIA Tesla K80 GPU [14] and an Intel Haswell CPU, and 30-80$\times$ as efficient [206].

2) Software developments

To compliment hardware acceleration in deep learning, various frameworks and software libraries have been built [156]. The main objective of any ML library is to facilitate quick design, prototyping and deploying of a deep learning model. Although now discontinued, *Theano* [270] was widely used to optimize and evaluate mathematical expressions for multi-dimensional arrays back in mid-2000s. Google's *TensorFlow* [44] is an open-source framework designed for training models at large scale in heterogeneous environments. It translates the entire model as a static computational graph comprised of nodes and edges. The nodes represent an operation (of type `tf.Operation`), while edges represent flow of data or a tensor (of type `tf.Tensor`) between different operations. Once a computation graph is created, it cannot be modified. Upon creation, it is fed to the TensorFlow execution engine, which performs further optimizations and launches computations via lazy loading, i.e., launches computations on nodes in the order of their dependency. Additionally, it comes with out-of-the-box support for CPU-GPU computations and supports a variety of programming languages like Python, Java, C++ and Javascript.

TF was inspired by a closed-source framework used internally by Google called *DistBelief* [91]. It supported data and model parallelism across thousands of CPU cores (and later GPUs as well). To address systems heterogeneity among worker replicas and handle replica failure, DistBelief supports data-parallelism via *Downpour SGD* and *Sandblaster*. Downpour SGD asynchronously pushes gradient updates to and from a central server coordinating server after a certain number of steps. Increasing the steps reduces the communication cost of aggregating updates from multiple model replicas. Since downpour SGD does not wait for updates from all workers at every iteration, it is more robust to node failures as training will not halt on account of failed replicas. Sandblaster is a framework that supports distributed batch optimization techniques like distributed L-BFGS (Limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm) [189]. It uses an external coordinator to divide work among model replicas as well as the central server shards holding model parameters.

MXNet [81] is a deep learning library that, like TensorFlow, also represents models as a dataflow graph and supports a variety of programming languages. The appeal of MXNet over TensorFlow lies in the fact that its programming model is both imperative as well as declarative. Declarative programming prioritizes optimization

over a clearly specified computation graph, while an imperative model offers more flexibility in designing new and custom models as well as model debugging. Additionally, MXNet comes with it own key-value store based implementation of a parameter server [168] for distributed training called KVStore. The store provides simple *set*/*get* operations for pushing/pulling model updates and also supports consistency models like sequential and eventual consistency.

Caffe [143], short for Convolutional Architecture for Fast Feature Embedding, was developed by Berkeley AI Research (BAIR) to have a declarative, fast and modular deep learning framework. Following, *Caffe2* [5] was developed by Facebook (now Meta Inc.) as a developer-friendly framework for large-scale model deployment. It comes with Python and C++ bindings and compatible to be run across multiple mobile operating systems and even small SoCs (System-on-chips) like Raspberry Pi. Caffe2 provides AllReduce-based decentralized communication via libraries like Message Parsing Interface (MPI) and NVIDIA Collective Communication Library (NCCL) [31] for distributed training across multiple CPUs or GPUs.

Formerly Meta's *PyTorch* [210] (now part of the Linux foundation), initially based on Lua-based library Torch [24], was later integrated with Caffe2 to provide a unified framework for deep learning. Out of the box, PyTorch comes with a distributed training module [170], tools for model deployment and serving [26], API for mobile development [25] and cloud support. PyTorch supports dynamic computational graphs that can be modified at runtime, and provides three levels of abstraction: *Tensor*, *Variable* and *Module*. A *Tensor* is an imperative n-dimensional array to be executed on a core, while a *Variable* is a node in the computation graph that stores the data and gradient. The layers of a neural network written in PyTorch are abstracted under a *Module* which stores the model state.

Petuum [301] was developed with the target of big data and large model training. It thus supports data and model parallelism to scale-up/scale-out distributed training. Petuum uses a Parameter server approach with stale synchronicity for communication-optimal distributed training. Due to stale synchronous communication approach, final model quality may not be as good as fully-synchronous training with Parameter servers or AllReduce-based aggregation. It also comes with default support for Apache Hadoop and HDFS [251]. To enable model-parallelism in Petuum, developers need to implement the *schedule* and *pull* method in order to schedule computation and aggregation on partial parameters among worker replicas.

CNTK [240], Microsoft's Cognitive Toolkit made two key contributions: it implements compression via *1-bit SGD* that quantizes gradients to 1-bit. This considerably reduces communication cost of aggregating gradients among all worker replicas. CNTK also proposes *Block-momentum SGD* that divides training data into multiple blocks and splits. Each worker trains a split on each block and computes gradients across all splits within a block. The gradients are then averaged to get weights for the block. Finally, block updates are fused into a global model such that the SGD update uses a block-level momentum and learning rate schedule. In addition, CNTK supports automatic hyperparameter tuning for training space exploration, comes with optimized dataloaders for massive training datasets, and provides high as well as low level Application Programming Interface (API) in Python, C++, C# and Brainscript. Another deep learning kit from Microsoft called Distributed Machine Learning Toolkit (*DMTK*) [10] was built to be used on top of CNTK. It comes with a custom Parameter server implementation called *Multiverso* for asynchronous communication patterns.

MLlib [195] is a library built for training models over the big data stack and integrated into Apache Spark [312]. *Keras* [85] is a high-level library that uses Tensorflow, Theano or CNTK as its backend. It comes with a scikit-learn type API and enables quick prototyping of DNNs, runs on both CPU and GPU, and completely modular in design.

Recent optimization libraries like *DeepSpeed* [4], [227], [289] enhance training and inference efficiency by combining compute, communication, memory and I/O. DeepSpeed minimizes memory footprint during training through the Zero-Redundancy optimizer (ZeRO) by partitioning the model and gradient states across multiple processes [221], uses mixed-precision training, supports a data-sampling pipeline for curriculum learning where simpler examples are presented during early training and difficulty is progressively increased [62], and enables training of Mixture-of-experts (MoE) models [145] where multiple networks contribute towards the final output through a gated network that helps redirect the input towards different experts. Additionally, it supports various parallelization strategies such as data parallel, model parallel, pipeline parallel as well as their combinations (different parallel schemes are discussed in greater detail in §2-B).

PyTorch FSDP module (fully-sharded data-parallel) [324] shards across multiple devices or nodes for fast training of large DNNs, particularly ones that that too big to fit on a single device. FSDP additionally performs gradient

sharding and optimizes communication by exchanging only the necessary shards and via mixed-precision training.

B. Parallelization strategies for Distributed Deep Learning

Training a neural network is an iterative-convergent process that requires repeatedly feeding a model with large amounts of data until the loss between model prediction and ground truth reaches a low enough target. As mentioned in §1-B, it is necessary to distribute deep learning models across multiple nodes/devices either when: (i) a model is too big to be accomodated on a single node. (ii) the training data is huge or the duration of training (i.e., iterations or epochs) is too high for the model to run sequentially on a solo machine.

Based on the type of partitioning and the granularity of parallelization applied on a model, there are three broadly defined classes of parallelism for distributed training of DL models [284], [265], [162]:

- *Data Parallelism*: Here, a model is replicated across multiple workers such that each worker trains independently on a unique subset of data [244]. The model then computes gradients from unique mini-batches on the individual workers. In case of synchronous data-parallel training [230], the gradient updates are exchanged among workers, followed by applying SGD update using the aggregated gradients to adjust model parameters. In contrast, local gradients could first be applied to the local model parameters, followed by parameter synchronization among workers. Both gradient and parameter aggregation strategies are equivalent in fully synchronous training as long as the initial model state prior training was identical among workers, data is sampled in an IID manner and synchronization operation is called at the end of every iteration. Depending on the topology, there are asynchronous [178] and partially-synchronous [132] data-parallel training algorithms as well. Data-parallel training is supported by most deep learning frameworks like deeplearning4j, MLlib, Caffe2, CNTK, Tensorflow, PyTorch, MXNet, Petuum [48], [195], [5], [240], [43], [301], [170], [81].

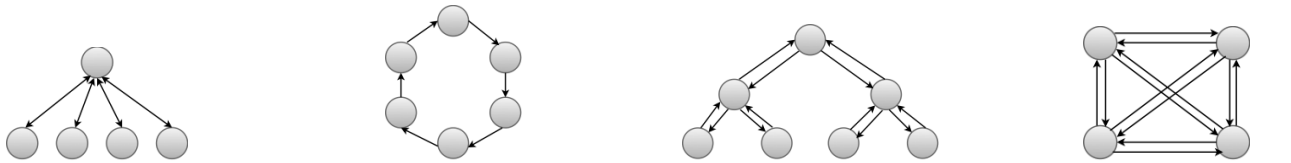
- *Model Parallelism*: We partition the model itself instead of training data among participating workers in this approach. This means that there is a single global model split across a cluster of nodes such that each worker operates on a small part of the entire model. Model-parallelism is necessary for training when a model cannot be loaded into memory of a single device [72], [250]. This strategy is also useful in inference tasks and model serving, especially when device memory is limited and request-rate is low [177]. It can be implemented by partitioning a model *horizontally* or *vertically*. Vertical partitioning involves splitting a model by the layers, while horizontal partitioning splits model parameters across multiple layers. It is crucial to design optimal partitioning schemes as model-parallel tends to be communication-heavy. Communication is required in the forward pass to communicate intermediate data between workers, while it is necessary to transmit gradients and activation maps in the backward pass. Further, naively splitting layers can lead to unbalanced parameter sizes on some workers leading to computational stragglers or network saturation on account of frequent communication of large tensors. Although majority of DL frameworks support data-parallelism, not all libraries support model-parallelism out of the box as it generally requires non-trivial code integration that may be specific to the architecture of a given model. Some of the frameworks that trivially support model-parallelism include TensorFlow, PyTorch, MXNet and Petuum. Model-parallel can further be branched into *pipeline* and *tensor-parallelism*, described as follows:
  - *Pipeline-Parallel*: Basic model-parallelism suffers from poor device utilization since only one worker is active at a time. Pipeline-parallel adds to model-parallelism by splitting training mini-batch into multiple micro-batches and pipelines their execution across workers. The idea was first proposed in *GPipe* [137] with the goal of partitioning models to better utilize cluster hardware and maximize application throughput.

  - *Tensor-Parallel*: Here, the computations within a single layer, such as feature maps, gradients and error tensors are split across multiple nodes or devices. This way, each device handles a subset of computations performed for a single layer, i.e., intra-layer parallelism [287].

- *Hybrid Parallelism*: Various solutions have been proposed combining different parallelization strategies either for more optimal scaling of deep learning models or improving overall throughput/reducing training time. Examples include GPipe [137] that combines model-parallelism with pipeline-parallelism. Another framework called PipeDream [201] combines data, model and pipeline-parallelism. Compared to synchronous data-parallel

training, PipeDream has considerably lower communication overhead as only partial tensors from a layer are communicated instead of the entire model gradients. Additionally, pipelining the input with micro-batch training allows complete overlap of computation and communication. Another fusion of model with pipeline-parallelism is NASPipe [322] that optimizes the training of a supernet as part of its neural architecture search (NAS). FlexFlow [144] evaluates different parallelization strategies with a Markov Chain Monte Carlo (MCMC) search algorithm. Works like HyPar [258] combines data, model and tensor-parallelism and partitions feature maps, kernels, gradients and error tensors to minimize overall communication cost, while AccPar [257] further optimizes the cost model to take computation into account over heterogeneous accelerator arrays. ZeRO and DeepSpeed [221] combine data and model-parallelism to overcome the limitations of each; high memory footprint of data-parallelism and poor scaling of model-parallelism due to fine-grained computation and expensive communication.

C. Training under Different Cluster Topologies



(a) Centralized Star (b) Decentralized Ring (c) Decentralized Tree (d) Decentralized P2P

Fig. 1: Basic network topologies in distributed computing.

Besides the parallelization scheme used for training, the arrangement of the physical nodes and devices within the cluster, i.e., the *topology* is another aspect to consider in distributed deep learning. As different workers/devices perform forward-backward pass on their local training data and model replicas, the updates at the end of each iteration are dictated by the particular communication pattern chosen in a given setting. The topology of a cluster determines the degree of distribution as well as the latency between the individual nodes for synchronizing model updates. Thus, the overall throughput of training a model can vary from topology to topology given a particular communication pattern (synchronous, asynchronous, stale-synchronous, etc.). A network topology can be classified as *centralized* or *decentralized* [61] based on the physical node arrangement.

- *Centralized systems*: This type of topology assumes a central location where the global copy of the model is stored or all the individual updates from workers are sent to for centralized processing (like applying SGD update). Thus, a centralized topology follows a hierarchical approach to aggregation where the reduction operation may be performed on the computed gradients or model parameters depending on the synchronization mechanism. A *Parameter Server (PS)* [168] strategy running a single parameter server and multiple workers training concurrently on different batches of data is an example of a centralized topology. At the beginning of every iteration, workers *pull* global parameters from the PS and compute gradients during backpropagation. The updates from the workers are then *pushed* to the PS which averages all the gradients and updates model parameters before proceeding to the next iteration. In *Ensemble* learning [112], multiple models are combined to produce an averaged model with better generalization performance. Workers train a model locally, which is then sent to a centralized location to produce an aggregated model. For a non-convex problem running for T iterations across N workers, centralized parallel SGD has been show to have a convergence rate of $\mathcal{O}(1/\sqrt{TN})$ [49], [92], [179]. The communication complexity is proportional to the number of workers, $\mathcal{O}(N)$, while the computational complexity to get a ϵ-approximate solution is $\mathcal{O}(N/\epsilon + 1/\epsilon^2)$ [181]. Figure (1a) shows a centralized system based on star topology where one node is connected to all other nodes in the system.

- *Decentralized systems*: Unlike centralized topology, there is no single node in decentralized systems that serves as the solo point of aggregation. Each worker keeps a local model replica and may only communicate with its neighbors. A *tree* or a *ring* topology is such an examples of a decentralized system. A node can only communicate with its parents and children nodes in a tree topology. In a ring topology, a node can

communicate only with its left and right neighbor. A PS strategy will multiple parameter server shards such that each holds a subset of the global parameters [146] can be considered decentralized as not a single node can be a potential single point of failure (SPOF). A *peer-to-peer* network has every node is connected to every other node. Such a topology has high scalability as workers can directly communicate with each other and the network will not get saturated due to heavy traffic to/from a single or a subset of nodes. [181] also demonstrated that the convergence rate in decentralized systems is $\mathcal{O}(1/T + 1/\sqrt{TN})$, if T is large enough. Although the computational complexity is decentralized training is the same as centralized, i.e., $\mathcal{O}(N/\epsilon + 1/\epsilon^2)$ [181], the communication complexity is considerably lower, equal to the degree of the network $\mathcal{O}$(Degree(network)). *Gossip-based* architectures are another example of decentralized systems where a worker communicates with a subset of workers. A challenge in gossip-based algorithms is to attain consensus, i.e., identical model state among all workers in the cluster. Communication in decentralized systems is facilitated via collectives and point-to-point (P2P) primitives like AllReduce, Broadcast, Scatter, Reduce, iSend and iRecv. Libraries like OpenMPI [17], NCCL [31] and Gloo [6] provide efficient and fast implementation of these collective operations such as Ring-AllReduce, Tree-base AllReduce and Recursive Doubling [211], [106]. Faster interconnects (PCIe gen-4 with 16 GT/s, gen 5 with 32 GT/s and NVLinks [167]) and high speed NiCs (network interface cards) on multi-GPU systems have made decentralized systems even more popular over the years [281]. Open-source libraries like *Horovod* [242] implement efficient inter-GPU communication via ring reduction. It is compatible with both PyTorch and TensorFlow and can be integrated in only a few lines of code such that it uses MPI for multi-CPU, and NCCL for NVIDIA-based multi-GPU communication. Figure (1b)-(1d) shows decentralized systems in ring, tree and peer-to-peer (P2P) topology respectively.

D. Deep Learning with Model-Parallelism

Efficient model-parallel partitioning schemes are more critical than ever with the growing size of DL model (read: large language models). These strategies are also necessary when training or deploying models on the edge where devices have relatively lower compute resources (i.e., CPU, memory and storage) compared to a typical HPC cluster or the cloud. Naively partitioning a model can even lead to failures like OOM (out-of-memory) error when devices in the cluster suffer from systems heterogeneity. An example of systems heterogeneity would be a 2 GPU cluster where one is a Tesla K80 with 11 GB VRAM and the other is an Ampere A100 with 40 GB VRAM. Splitting a hypothetical model with 50 GB worth of parameters evenly into half (i.e., 25GB each) among the GPUs will not work as the smaller device cannot accommodate its partitioned share. Thus, model-parallel training warrants intuitive partitioning schemes. The communication overhead is another important consideration with this approach. Model-parallel training needs to communicate its intermediate data among workers that hold consecutive layers of a model. For devices n_1 and n_2 holding model layer i and $(i+1)$ respectively, intermediate data needs to be sent from layer i to $(i+1)$ during forward pass between n_1 to n_2. During backpropagation, gradients and activation maps are communicated from $(i+1)$ to i between n_2 to n_1. For a model with L layers, the total communication time t_{comm} at the end of an iteration in model-parallel is given as shown in Equation (2).

$$t_{comm} = \sum_{i=1}^{L}(\alpha\lceil log(P)\rceil + \beta B\frac{N-1}{N}d_i) + 2\sum_{i=2}^{L}(\alpha\lceil log(P)\rceil + \beta B\frac{N-1}{N}d_{i-1}) \tag{2}$$

Here, α and β refer to the latency and inverse bandwidth in accord with the α-β communication model, N is the number of workers, L is the number of DNN layers and B is the mini-batch size. The first sum is the time taken by all-gather required after every layer, while the second sum accounts for the all-reduce time for transmitting gradients during backpropagation. Here, we used ring-allreduce and Bruck's algorithm for all-gather [269].

- *AMP*: Algorithms like *asynchronous model-parallel* [114] are specially designed to train on networks of interconnected devices like GPUs. Gradients are computed and accumulated on each device until it reaches a certain threshold, following which those updates are applied and the accumulator buffer is cleared. The local update is executed in an asynchronous manner, i.e., without any synchronization with neighboring DNN layers across other workers. The accumulation/update frequency is user-specified; a small value increases gradient staleness while a large value reduces gradient variance at the cost of infrequent updates and thus, slowing down model convergence.

- *SpecTrain*: [79] proposes pipelined model-parallelism with high GPU utilization via a weight prediction technique. DL training with pipeline processing ensures high device utilization by constantly processing mini-batches in the worker pipeline, but suffers from gradient staleness and weight consistency issues. The staleness comes from the later mini-batches in the pipeline begin processing before the previous mini-batches have successfully updated model weights. SpecTrain predicts future weights to perform computations for the current mini-batch instead of using stale parameters. The key idea here is that smoothened gradients in momentum SGD [204] reflect the trend of parameter updates. For a decay factor γ and gradients g_t computed at iteration t, momentum SGD is given by $v_t = \gamma \cdot v_{t-1} + (1-\gamma) \cdot g_t$. We replace the actual gradients g_t with smoothened gradient v_t to perform SGD update to compute predicted weights such that $\widehat{W_t} = W_t - \eta \cdot v_{t-1}$. For a specific version of weights and version difference s, SpecTrain recursively applies its modified SGD update equation as $\widehat{W_t} = W_t - s \cdot \eta \cdot v_{t-1}$.

- *Alpa*: Frameworks like *Alpa* [326] fuse data, model and pipeline parallelism. It further classifies model-parallelism as inter-operator and intra-operator parallelism. In intra-op parallelism, the individual layers or operators of a model are partitioned among workers, while inter-op parallelism partitions the computational graph itself. A *sharding spec* is used to define the layout of a tensor. For example, a matrix or a 2-dimensional tensor X_0X_1 can be sharded such that $X_i = S$ implies the i-th axis of the tensor is partitioned. Thus, SX_1 means the tensor is row-partitioned, X_0S implies it is column-partitioned, SS means both row and column are partitioned and X_0X_1 implies it was replicated without any partitioning.

- *Cross-mesh resharding*: [328] is a communication pattern for large-scale model parallelism based on inter and intra-operator parallelism. This is a many-to-many multicast communication problem where a sharded tensor is sent from a source to a destination device mesh. To address this issue, [328] proposes an efficient broadcast communication mechanism and an overlapping-friendly pipeline schedule.

- *Diversely Stale Parameters (DSP)* : *Layer-wise Staleness* and DSP [302] deal with straggler slowdown in model-parallel training. Sequential propagation of activations and error gradients suffers from *backward* and *forward locking* due to computation dependency between consecutive layers. The *update* stage is also locked as backward pass will not begin until forward pass has fully executed. Layer-wise staleness allows different layers of a model to be trained independently with stale parameters, thus removing the synchronization barrier in model-parallelism. Based on this idea, the proposed DSP algorithm trains a model such that lower layers use more stale information than upper layers for parameter update. To reduce memory footprint, it also incorporates checkpointing and recomputation [120]. The authors provide a convergence analysis for DSP which is proved to converge to critical points for non-convex problems with SGD and momentum SGD.

E. Training with Hybrid Parallelization Schemes

Data-parallelism assumes the entire copy of the model and intermediate can fit onto the memory of a single device. Although model-parallelism partitions a model at varying granularity (i.e., inter and intra-op parallelism), it still suffers from poor device utilization as devices have to sit idle in sequential components of forward-backward pass. To alleviate this problem and improve overall system utilization, pipeline parallelism splits a mini-batch into micro-batches and keeps the execution pipeline full at all times, while tensor parallelism leverage intra-layer parallelism to split the individual layer of a model across many workers. The most effective training methodologies combine pipeline, tensor and hybrid parallelism for large model training, few of which are described here:

- *PipeDream*: [201] partitioned a model into multiple stages where each stage residing on one device is a collection of a consecutive set of model layers. To ensure high GPU utilization, multiple mini-batches are fed to the training pipeline, thus combining data, model and pipeline parallelism. After forward pass, each stage asynchronously sends its output activations to the next stage while concurrently processing a new mini-batch. Upon completion of backward pass, each stage similarly communicates its gradients asynchronously to the previous stage while computing a new mini-batch. Asynchronously transmitting output activations or gradients in-between stages results in better hardware efficiency due to significant overlap of computation and communication. The communication overhead itself in PipeDream is lower than conventional data-parallel

training as the intermediate data and gradients communicated for any stage are significantly lower than the entire parameters exchanged in data-parallel. To efficiently partition layers across workers, PipeDream also implemented a profiling mechanism to measure for every layer its total computation time across forward-backward pass, the size of output activations and backward gradients for the layer, and the parameter size for the given layer. Using these metrics, the partitioning algorithm uses dynamic programming to output a partitioning of layers into stages, replication factor required for each stage, and the number of active mini-batches (NOAM) needed to keep the training pipeline busy. Every worker in pipelined training can either perform forward work by processing a new mini-batch and traverse from initial to final layer stages, or perform backward pass of computing gradients for current mini-batch. PipeDream uses *one-forward-one-backward (1F1B)* scheduling mechanism where each stage alternates between performing forward and backward pass for a mini-batch. To deal with parameter staleness, PipeDream uses *weight stashing* and *vertical sync*. The former keeps a separate versions of model parameters for each active mini-batch. Thus, we use the same copy of weights for both forward and backward pass of a specific mini-batch and ensure consistency in weight updates. Vertical sync eliminates inconsistency across stages where the latest version of model weights is used for computing activations and gradients based on the active mini-batch.

- *GPipe*: [137] proposed a batch-splitting pipeline that resulted in near-linear speedup on a multi-GPU setup. Similar to PipeDream, a model can be partitioned into a sequence of layers such that consecutive layers are placed in a common cell that is placed on a worker. Instead of a sequence of active mini-batches in the training pipeline, GPipe splits a mini-batch itself into *micro-batches* and pipelines its execution across partitioned cells. The gradients over all the micro-batches are accumulated synchronously before applying the SGD update. Thus, updates are consistent regardless of the total partitions. Compared to data-parallel training, the communication cost of GPipe is lower as well since only output activations or gradients corresponding to a cell of layers needs to be communicated rather than the entire model parameters.

- *HyPar*: [258] is a hybrid parallelism technique that combines data, model and tensor-parallelism to split the input and output feature maps, gradients, kernel tensors as well as error tensors across every layer of a DNN. Given a cluster of nodes, HyPar partitions a model in linear time using a hierarchical layerwise dynamic programming method. The objective of the partitioning scheme is to minimize the overall communication cost in distributed DL. Compared to model-parallel, this approach significantly reduces the cumulative communication overhead, and gives significant performance and efficiency gain over data-parallelism. An extension of this work, called *AccPar* [257] further optimizes for distributed training over heterogeneous workers by modifying the cost model to include computation and communication in order to generate the layer-partitioning scheme.

- *PTD-P* [202] combines pipeline, tensor and data-parallelism to train large language models with good application throughput. It implements pipeline parallelism across multi-GPU servers, tensor-parallelism within a multi-GPU server, and data-parallelism to train up to trillion parameter models. With data and model parallelism, the former should be used to scale-up training to more devices, while the latter should be used such that its model parameters and intermediate metadata can fit in the device memory. For pipeline-parallelism, the optimal microbatch-size depends on the overall throughput and memory footprint characteristics of the model, along with the pipeline depth, degree of replicated model state (i.e, data-parallelism) and overall batch-size.

- *NASPipe*: [322] proposed pipelined training of a supernet (i.e., a collection of DNNs with different architectures and hyperparameter configurations that are trained collectively). To reduce GPU memory footprint and improve execution efficiency, NASPipe uses a context switching approach where it moves the whole supernet to CPU memory, predicts a subnet's schedule and loads or evicts a subnet before or after its execution. It performs supernet training via causal synchronous parallel pipelining such that the supernet is partitioned across workers and concurrently executed as a collection of various subnets. The causal dependency in a supernet arises from different subnets working on the same set of layers. If subnet s_1 performs computation with layer l, followed by subnet s_2 working on l as well, then s_2 has a causal dependency on s_1 because s_2 should read parameters of l in its forward pass with the update made in the backward pass of s_1. NASPipe parallelizes supernet training with inter-subnet parallel task generation and pipeline parallelism. For the former, it launches multiple subnets

concurrently such that each subnet processes one input batch. Each subnet is the partitioned into multiple stages such that one device holds a single stage and forms parallel subnet executions into a pipeline. NASPipe has various processes as part of its execution: a *scheduler* that observs tasks in forward and backward pass queues and checks whether it satisfies dependency preservation of its CSP (causal synchronous parallel) pipeline, a *predictor* that forecast upcoming tasks with maximum likelihood to be scheduled in coming steps, a *context executor* that iteratively fetches context schedule from CSP and executes subnets, and a *context manager* that shares the supernet context with the context executor for completely asynchronous management. NASPipe achieved upto 7.8× better performance than baselines for supernet training even with causal dependencies among subnets.

- *HetPipe*: [209] is a distributed DL system that integrates pipelined model-parallelism (PMP) with data-parallelism (DP) on heterogeneous GPU clusters. A group of GPUs called a *virtual worker* trains on mini-batches in a pipelined manner. Multiple virtual workers are deployed for data-parallel training. HetPipe synchronizes updates among these virtual workers (doing both PMP and DP) with Wave Synchronous Parallel (WSP) communication. WSP extends the stale-synchronous parallel communication model [132] to include PMP and DP training for multiple virtual workers. Each virtual worker suffers from *local* and *global staleness* as it keeps a local copy of model parameters that it synchronizes infrequently. Local staleness arises due to pipelined mini-batches that may not reflect the current weights as computed by the last mini-batch. Global staleness comes from WSP in an attempt to reduce the communication overhead between virtual workers and the parameter server. A worker pushes aggregated updates from a wave (i.e., a sequence of mini-batches processed concurrently in a virtual worker) instead of pushing updates of every mini-batch to the parameter server. Compared to baseline synchronous data-parallel training, HetPipe converged upto 49% on state-of-the-art DL models.

3. Training Algorithms and Communication Patterns in Deep Learning

A. Overview of Distributed Training Techniques

A plethora of distributed learning algorithms and SGD variants have been proposed for training DNNs. The communication patterns followed by these algorithms may either be optimal for a cluster of nodes arranged in a centralized or decentralized topology. We describe some significant distributed DL algorithms in this section.

a) Bulk-Synchronous Parallel (BSP):

In bulk-synchronous data-parallel training, multiple workers collaboratively train a model such that each worker processes a unique partition of training data. Once each worker computes gradients over a data partition with its local model replica, the updates are aggregated together before applying SGD update to adjust model parameters in the direction of minima. The updates from various workers are synchronized between consecutive iterations. A model with weights w_t training on N workers processes a mini-batch $x_{(n,t)}$ of size b from dataset $\mathcal{B}_n$ on worker n in order to optimize loss function $\mathcal{L}$ as per Equation (3).

$$w_{t+1} = w_t - \eta \frac{1}{N} \sum_{n=1}^{N} \frac{1}{|b|} \sum_{x_{(n,t)} \in \mathcal{B}_n} \frac{\partial}{\partial w_t} \mathcal{L}(x_{(n,t)}, w_t) \tag{3}$$

Parameters at iteration $(t+1)$ are computed from those at t and updated in a direction opposite to the gradients, scaled by learning rate η. The first summation implies how the gradients are accumulated from N workers and then averaged together. We generally consider *weak-scaling* in synchronous training; the per-worker batch-size is fixed and more workers can be added to the cluster to process proportionally more samples per-training iteration (i.e., higher mini-batch size). This results in fewer iterations to complete a single pass or epoch over training data; training a model for 'E' epochs over dataset-size 'D' with batch-size 'B' results in training steps $T = \left(\frac{ED}{B}\right)$.

Accumulating updates for large models incurs high communication cost in BSP training. Since the synchronization step is blocking, BSP training typically suffers from straggler slowdowns [274], [276], especially in heterogeneous clusters. For non-convex problems like DNN training, BSP has a convergence rate of $\mathcal{O}(1/\sqrt{NT})$ for N concurrent workers training to T iterations.

TABLE I: Latency-bandwidth cost of common collectives in distributed training

Operation	Latency Complexity	Bandwidth Complexity	Total communication cost
Parameter Server (Star topology)	$\mathcal{O}(1)$	$\mathcal{O}(MN)$	$2\alpha + 2(N-1)M\beta$
Ring-Allreduce	$\mathcal{O}(N)$	$\mathcal{O}(M)$	$2\alpha(N-1) + 2\frac{(N-1)}{N}M\beta$
Tree-Allreduce (pipelined)	$\mathcal{O}(\log(N))$	$\mathcal{O}(M)$	$2\alpha\log(N) + 2M\beta$
Recursive Having and Doubling	$\mathcal{O}(\log(N))$	$\mathcal{O}(M)$	$2\alpha\log(N) + 2\frac{(N-1)}{N}M\beta$
Rabenseifner-Allreduce	$\mathcal{O}(\log(N))$	$\mathcal{O}(M)$	$2\alpha\log(N) + 2\frac{(N-1)}{N}M\beta$
Reduce-Broadcast	$\mathcal{O}(N)$	$\mathcal{O}(M\log(N))$	$2\alpha\log(N) + 2\log(N)M\beta$

BSP training may be centralized or decentralized depending on the chosen communication pattern. Centralized synchronous training is achieved via *parameter servers* [168], [146] that collect gradients from all workers, average them, apply SGD update and finally redistribute updated model parameter back to the workers to resume next iteration. In terms of the α-β communication model, the communication cost of parameter server training is $\mathbf{2}\alpha + \mathbf{2M}\beta(\mathbf{N-1})$, where α is the latency, β is inverse of network bandwidth and M is the model size.

In decentralized BSP, there is no central point of aggregation and the workers aggregate their gradient updates using collective communication or point-to-point primitives [17], [31], [6]. AllReduce is the primary communication operation used for gradient reduction, with specific implementations like Ring-Allreduce, Tree-Allreduce and Recursive Doubling [51], [281], [211], [106], [269].

In *Ring-Allreduce*, a tensor is divided evenly among the workers and each worker communicates its chunk to the right neighbor while receiving the chunk from its left neighbor. For a model of size M, each worker thus communicates M/N elements to the right neighbor $(N-1)$ times. Once each chunk is reduced on worker, a worker sends the reduced chunk over, incurring additional overhead of $M(N-1)/N$. The total communication cost for Ring-AllReduce is thus $\mathbf{2(N-1)}\alpha + \mathbf{2M}\beta\mathbf{(N-1)/N}$. Ring-Allreduce is bandwidth-optimal compared to general allreduce where all data is sent across all workers, incurring cost $\mathbf{2(N-1)}\alpha + \mathbf{2M}\beta\mathbf{(N-1)}$.

Tree-Allreduce with pipelining is based on double binary trees [238] that combine full bandwidth for broadcast and reduce operations. Double binary trees rely on the fact that half or less ranks in a binary tree are nodes while half or more ranks are leaves. Data is sent across the tree hierarchy in pairs of two across multiple rounds, $\log(N)$ to be exact, thus offering logarithmic latency. The α-β communication cost of tree-based allreduce is $\mathbf{2}\alpha\log(\mathbf{N}) + \mathbf{2M}\beta$.

In *Recursive Doubling* [269], communication is carried out multiple steps. In the first step, workers immediately next to each other in the topology exchange their data. In the next step, workers exchange data with even further nodes such that all processes communicate all of their data in $\log(N)$ steps. Each rank thus communicates M/N data in the first step, $2M/N$ data in the second step and $2^{\log(N)-1}M/N$ data in the last step. The total communication cost of allreduce in recursive doubling is $\alpha\log(\mathbf{N}) + \mathbf{M}\beta\log(\mathbf{N})$.

Table (I) describes the latency and bandwidth complexities of different versions of allreduce, such as ring, tree, reduce-broadcast, recursive halving and doubling and rabenseifner. Communication cost of the last two are identical as they're the same; rabenseifner allreduce involves two steps: reduce-scatter via recursive halving, and allgather via recursive doubling. In-depth details of different aggregation collectives can be found in [216], [269]. With faster generations of PCI-e (peripheral component interconnect express) lanes and development of high-speed interconnects like NVLinks, allreduce-based decentralized training has become extremely popular over the years. Further algorithmic optimizations in low-latency and bandwidth-optimal allreduce have furthed helped its cause. DL Benchmarks like DawnBench [87] reported that majority of the submissions used allreduce for distributed synchronous training.

b) Asynchronous Parallel (ASP):

In asynchronous parallel training [169], devices train independently in a non-blocking manner, i.e., there is no synchronization barrier. Systems like Hogwild! [205] inspired asynchronous training by implementing distributed SGD in a lock-free manner on shared memory systems. In ASP, workers push/pull their gradient updates to/from the parameter server asynchronously, akin to Downpour SGD [91]. The difference is that the global model state is preserved on the PS in ASP training, which applies SGD update based on the gradients received from a single worker. Since only one worker updates the global model at the PS at a given iteration, the SGD update on the PS

with ASP is performed as shown by Equation (4).

$$w_{t+1} = w_t - \eta \frac{1}{|b|} \sum_{x_{(n,t)} \in \mathcal{B}_n} \frac{\partial}{\partial w_t} \mathcal{L}(x_{(n,t)}, w_{n,(t-\tau_{n,t})}) \ \ \forall \ \ n \in [\, 1, 2, 3, ...N \,] \tag{4}$$

Here, N workers train asynchronously such that worker n computes gradients on mini-batch sample $x_{(n,t)}$ with stale parameters $w_{n,(t-\tau_{n,t})}$ such that weight on worker n is delayed from the current iteration t by a factor of $\tau_{n,t}$. The gradients thus computed can be inexact if τ is large as the parameters used during backpropagation will be stale in that case.

Convergence in ASP training suffers on account of staleness where slower workers still train with parameters several iterations behind the latest model state and thus, produce stale updates [276]. Due to this, model progress is thrown back as the PS receives stale gradients from the slower workers applies it to update model parameters that are then dispersed to other workers in the cluster during the "pull" operation. In a homogeneous cluster setting where stragglers are not an issue, centralized ASP training has a convergence rate of $\mathcal{O}(1/\sqrt{NT})$. In decentralized ASP [182], [180], worker updates are aggregated via symmetric peer-to-peer communication such that each active worker sends its updates to a passive worker and waits for the passive worker to send back its parameters. Decentralized ASP has a theoretical convergence rate of $\mathcal{O}(1/\sqrt{T})$.

Other asynchronous training techniques like [121] implement sparsified upward communication from worker to PS for further efficiency. Sparse communication is performed randomly via uniformly sampling local updates corresponding to a worker. This approach converged linearly in strongly convex optimization and was also applicable to automatic dimensionality reduction. [197] correlates asynchronicity to adding momentum implicitly in distributed SGD. They observed that in settings with both implicit and explicit momentum in SGD, negative values of explicit momentum are optimal for training.

The communication cost of ASP is significantly lower than BSP training. However, this comes at the cost of model inconsistency across the workers due to the asynchronous nature of ASP and difficulty to achieve convergence, especially in heterogeneous clusters. By design, ASP performs less work per-iteration than BSP since the former trains on a mini-batch of size b while the latter trains with a global batch-size of Nb combined over all the participating workers. If staleness is considerable, ASP may achieve lower convergence than BSP or may require many more iterations to attain the same convergence target.

c) Stale-Synchronous Parallel (SSP):

$$\begin{aligned} w_{n,t+1} = w_0 - \eta \sum_{i=1}^{t-s} \sum_{j=1}^{N} \frac{1}{|b|} \sum_{x_{(j,i)} \in \mathcal{B}_n} \frac{\partial}{\partial w_{j,i}} \mathcal{L}(x_{(j,i)}, w_{j,i}) - \eta \sum_{i=t-s}^{t} \frac{1}{|b|} \sum_{x_{(n,i)} \in \mathcal{B}_n} \frac{\partial}{\partial w_{n,i}} \mathcal{L}(x_{(n,i)}, w_{n,i}) \\ - \eta \sum_{(j,i) \in \mathcal{S}_{n,t+1}} \frac{1}{|b|} \sum_{x_{(j,i)} \in \mathcal{B}_n} \frac{\partial}{\partial w_{j,i}} \mathcal{L}(x_{(n,i)}, w_{j,i}) \end{aligned} \tag{5}$$

SSP training is an intermediate approach between synchronous and asynchronous training strategies. Workers are allowed to train asynchronously in stale-synchronous training, but only to a certain extent [132]. This approach performs especially well in heterogeneous clusters which inherently suffer from straggler slowdowns. Here, faster workers can make more progress than slower workers, but bounded after a certain point and denoted as the staleness threshold. This threshold measures how much stale parameters can send update the global model state on the parameter server. Once the staleness threshold is crossed, faster workers halt training until the slower workers catch up. Due to bounded asynchronicity, the communication cost of SSP lies in-between ASP and BSP. Equation (5) elaborates the SGD update for stale-synchronous training for bounded staleness s [132], [265] on a worker n.

When the fast and slow workers are $\leq s$ iterations, all workers will see model updates in the range $[1, t-s]$ (the first summation in Equation (5). Additionally, there is an adaptive subset of model updates between iterations $[t-s, t]$. The per-worker computed gradients are noisy due to staleness in each worker's model state. $\mathcal{S}_{n,t+1}$ is a subset of updates other workers during the iteration phase $[t-s]$.

SSP enjoys a theoretical convergence rate of $\mathcal{O}(\sqrt{2(s+1)N/T})$ for N workers training with a staleness threshold s for T iterations [132], [154]. If $s = 0$, SSP is equivalent to BSP training [86], and tantamount to local training as $s \to \infty$.

Various other optimizations have been proposed on top of SSP. [123] uses value staleness to analyze the difference in model parameters and proposes *value-staleness-aware* SGD that uses the staleness parameter to train with dynamic learning rates on a per-iteration basis. The authors report training speedup of 1.26× and 75% error reduction compared to baseline SSP. *Dynamic SSP* or DSSP [323] is an extension of SSP on the parameter server framework that dynamically assesses the staleness threshold at runtime. A new threshold is chosen at every iteration based on a worker's latest batch processing time so as to reduce the waiting time on faster workers for synchronizing latest model updates. This increases the training throughput which speeds up the model convergence rate. *Free SSP* or FSSP [246] further optimizes SSP for homogeneous clusters where the performance of workers varies due to resource competition from concurrent jobs and high communication cost. FSSP alleviates the performance degradation due to slower nodes that bring down the overall training throughput. It counts the *slow times* or the number of times a slow worker made faster workers pause in SSP and classifies it as a straggler once it reaches *penalty times*. Upon identification of a straggler, the worker will be excluded from the training group and FSSP will copy its job to an available faster worker. This approach converged up to 12× faster than SSP and used 4× fewer iterations to converge than ASP.

d) Local-SGD:

In *Local-SGD*, workers train locally on data partitioned in an IID manner for certain time periods, followed by model synchronization among workers at particular intervals [262], [118]. This can be implemented in a centralized topology with PS, or in a decentralized setting calling allreduce for model aggregation.

$$w_{n,t+1} = \begin{cases} w_{n,t} - \eta \frac{1}{|b|} \sum_{x_{(n,t)} \in \mathcal{B}_n} \frac{\partial}{\partial w_t} \mathcal{L}(x_{(n,t)}, w_{n,t}) & \text{if } t \ \% \ T_s > 0 \\ w_{n,t} - \eta \frac{1}{N} \sum_{n=1}^{N} \frac{1}{|b|} \sum_{x_{(n,t)} \in \mathcal{B}_n} \frac{\partial}{\partial w_t} \mathcal{L}(x_{(n,t)}, w_t) & \text{if } t \ \% \ T_s = 0 \end{cases} \tag{6}$$

In Equation (6), each worker n trains on its local data $\mathcal{B}_n$ with local parameters $w_{n,t}$ for a specified number of iterations. After every T_s iterations, the updates from all workers are combined. The equation describes local-SGD for a decentralized setting where every worker may hold a different model state $w_{n,t+1}$ after gradient aggregation. When implemented with PS, the gradient updates are sent to a central server that updates the model that is consistent across all workers after pulling those updates from the PS. The weights at iteration $(t+1)$ thus become w_{t+1} instead of $w_{n,t+1}$.

When training with local-SGD for T iterations with N workers and using a synchronization threshold T_s, the convergence rate for DNN training approaches $\mathcal{O}(1/\sqrt{NT})$ as $T_s \to 0$ and $\mathcal{O}(1/\sqrt{T})$ as $T_s \to \infty$. This is because local-SGD generalizes to BSP as T_s gets smaller, while its equivalent to asynchronous training as T_s gets larger.

An extreme case of local-SGD called *One-shot SGD* trains models independently among various workers and simply averages all the local models at the end [329], akin to ensemble learning. In prior art, mini-batch SGD has been compared extensively to local-SGD [299] with the conclusion that the former is superior to the latter in heterogeneous settings. However, various adaptive versions of local-SGD have been proposed with better peak theoretical convergence. *Adacomm* [292] studied how convergence varies with the frequency of model averaging and developed an adaptive communication policy that begins with infrequent aggregation to minimize communication cost and gradually increases synchronization frequency to achieve models with lower error/higher accuracy. Adacomm reduces training time by up to 3× over BSP to achieve the same training loss. Local-SGD has also been studied as an alternative to large-batch training [186]. Although training with large-batches is more efficient as a model trains on more data on each iteration and also reduce the communication cost compared to small-batch training, it degrades the generalization performance (i.e., on unseen test data) of a DL model [152], [193], [119]. We explain this in greater detail in §3-C.

[186] proposed *post-local SGD* to improves generalization performance of large-batch training. In this approach, we first train with mini-batch SGD for certain iterations in the first phase, followed by local-SGD afterwards. During the mini-batch SGD phase, post-local SGD uses small local mini-batches and switches to larger mini-batches in the second phase (i.e., local-SGD). Such a strategy achieved better generalization than simply training with larger batches. Local updates with periodic averaging or LUPA-SGD [124] extended local-SGD to a theoretical convergence rate $\mathcal{O}(1/NT)$ for non-convex DNNs assuming the loss function satisfies Polyak-Łojasiewicz inequality and proposed an adaptive synchronization strategy for linear speedup.

Variance-reduced Local SGD (VRL-SGD) [184] applies to local training on heterogeneous or non-IID data settings. This approach improves the convergence rate on heterogeneous data by reducing the gradient variance among workers. VRL-SGD compares the deviation between the locally-computed gradients and the globally-averaged gradients. It then computes the stochastic approximation gradient which is computed by subtracting the deviation metric from the local gradients and apply that as part of the SGD update on each worker.

e) Gossip-based SGD:

$$w_{n,t+1} = w_{n,t} - \frac{\eta}{1+|\mathcal{F}(m)|} \cdot \{ \frac{1}{|b|} \sum_{x_{(n,t)} \in \mathcal{B}_n} \frac{\partial}{\partial_{w_{n,t}}} \mathcal{L}(x_{(n,t)}, w_{n,t}) + \mathcal{F}(m) \cdot \frac{1}{|b|} \sum_{x_{(m,t)} \in \mathcal{B}_m} \frac{\partial}{\partial w_{m,t}} \mathcal{L}(x_{(m,t)}, w_{m,t}) \}$$
$$\text{such that } \mathcal{F}(m) = p \cdot \mathbb{U}(1,N)(m) + (1-p) \cdot 0 \text{ and } \mathbb{U}(1,N)(m) = \begin{cases} 1/(N-1) & \text{for } 1 \leq m \leq N \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Gossip-based SGD (GoSGD) involves random walks across a network of nodes such that each worker exchanges its updates with only a subset of other workers in the cluster [110], [69], [265], [154]. In one-to-one GoSGD [69], each worker may decide to communicate its updates with probability p by choosing another worker from a uniform distribution, or train and update its model parameters locally.

While aggregating updates, communication happens via point-to-point primitives (e.g., send, recv, isend, irecv) instead of allreduce collective. Equation (7) describes a cluster of N workers where updates from worker n and m are aggregated depending on $\mathcal{F}(m)$ that chooses m with likelihood p from the probability density function $\mathbb{U}$ in the interval worker ranks in the cluster, $[1, N]$. When a worker is chosen, we average the updates for the two workers and thus, $|\mathcal{F}(m)| = 1$. When no worker is chosen for synchronizing updates, both $\mathcal{F}(m)$ and $|\mathcal{F}(m)| = 0$.

The convergence quality of GoSGD is determined by the communication probability, achieving a rate of $\mathcal{O}(1/\sqrt{2T})$ when $p \to 1$ and training to T iterations. This is a generalization of BSP training with updates aggregated among 2 workers. When $p \to 0$, GoSGD has a convergence rate of $\mathcal{O}(1/\sqrt{T})$ and is equivalent to local training. Additonally, GoSGD suffers from model inconsistency across workers due to the stochastic nature of worker selection for gradient reduction.

Gossip SGD with *periodic global averaging* or Gossip-PGA [83] which has a convergence rate of $\mathcal{O}(1/\sqrt{NT})$ when T is sufficiently large. In this algorithm, each worker collects updates from its neighbors during gossip-step, followed by global aggregation between all workers using allreduce primitives in the global-average step g. A small g incurs expensive communication among workers, while a large g reduces synchronization overhead by leaning more towards gossip-based SGD. When $g \to 1$, Gossip-PGA generalizes to BSP training and same as GoSGD as $g \to \infty$. A variant of Gossip-PGA called Gossip-AGA takes inspiration from [292] and uses *adaptive global averaging* periods to offer the same convergence guarantees. In Gossip-AGA, the global averaging period g is small during the early part of training until the variance among workers reduces substantially. Thus, synchronization cost is high initially since g is small. As iterations progress, variance gets smaller and g is gradually increased leading to communication savings with gossip-based training.

f) Elastic-Averaging SGD:

Elastic-Averaging or EA-SGD [317], [147] is a flavor of parameter server training that performs periodic communication between the PS and the workers. It has both synchronous and asynchronous variants. The intuition is that workers training locally for longer allows for more exploration of the workers into the local minima over the non-convex loss surface. The communication period T_s influences the communication cost and convergence of EA-SGD; $T_s \to 1$ implies PS-based BSP training. This is comparable to local-SGD, but applied to both asynchronous-synchronous PS setting where the deviation in model weights is compared between the local workers and the central server that holds the global model state. In other words, EA-SGD seeks to maximize the consensus between the local and central parameters. Thus, instead of minimizing just the loss $\mathcal{L}(\cdot)$, EA-SGD aims to minimize Equation (8a).

$$\min_{w_{(1,t)}, w_{(2,t)} \ldots w_{(N,t)}} \sum_{i=1}^{N} \frac{1}{|b|} \sum_{x_{(i,t)} \in \mathcal{B}_i} \mathbb{E}[\, \mathcal{L}(x_{(i,t)}, w_{i,t}) \,] + \frac{\rho}{2} ||w_{i,t} - w_{*,t}||^2 \quad \text{if } t \,\%\, T_s > 0 \tag{8a}$$

$$w_{n,t+1} = w_{n,t} - \eta \{ \frac{1}{|b|} \sum_{x_{(n,t)} \in \mathcal{B}_n} \frac{\partial}{\partial w_{n,t}} \mathcal{L}(x_{(n,t)}, w_{n,t}) + \rho \cdot (w_{n,t} - w_{*,t}) \} \quad \text{if } t \ \% \ T_s = 0 \tag{8b}$$

$$w_{*,t+1} = w_{*,t} + \eta \sum_{i=1}^{N} \rho(w_{i,t} - w_{*,t}) \tag{8c}$$

The penalty term ρ ensures that local workers do not fall into a minima that is too far from the global model state on the PS and thus lead to model inconsistency. Local and global updates (Equation (8b) and (8c) respectively) in EA-SGD are applied with the deviation between local and global updates, scaled by the penalty term ρ. During aggregation step after every T_s iterations, global model at the PS is updated by accumulating the deviation of all workers from the central parameters. The symmetric term $\eta\rho(w_{i,t} - w_{*,t})$ has the effect of an elastic force between the update of $w_{n,t}$ and $w_{*,t}$, and has a critical impact on the stability of the EA algorithm. With *momentum EA-SGD* (EAM-SGD), elastic averaging adds momentum to its SGD update mechanism.

B. Identifying and Optimizing Compute Cost

From a traditional HPC perspective, the overall computational throughput can be increased either by *scaling-up* or *scaling-out*, i.e., *vertical* or *horizontal* scaling respectively. In the latter, multiple devices or nodes collaboratively perform the training of a neural network. Data-parallel, model-parallel and tensor-parallel DL techniques fall under this category. In vertical scaling, computational performance is improved over a single training node, either by performing more work per-step or iteration, or by achieving a given task/operation with fewer instructions/cycles. Pipeline parallelism described in §2-B improves device utilization and thus, aligns itself as a vertical scaling technique. Additionally, development of specialized ML accelerators and ASICs, faster MMUs [206] and optimized kernel libraries [12], [288] improve the compute cost of forward and backpropagation, either by lowering the execution time or improving efficiency (i.e., lower power consumption).

Despite these developments, distributed training is not merely limited by communication and data-movement. 'Compute heterogeneity' is pervasive across cloud and shared clusters such that nodes or training devices may inherently vary in terms of their computational capability. Such heterogeneity also applies to training across different GPU architectures, mixed CPU-GPU training or in shared clusters where a subset of nodes are running concurrent jobs. It is even more apparent under federated settings, described in §5 in more detail. Under communication-heavy, synchronous training, compute heterogeneity suffers from stragglers where slower or inferior resources take longer time to process, causing faster resources to idle and wait until calculations are complete over the entire cluster [108], [274]. On the other hand, resource heterogeneity results in staleness with asynchronous training algorithms [205].

To ameliorate straggler slowdown, Anytime mini-batch [108] allocates a time-budget across a cluster, i.e., imposes a fixed time for each device to compute its updates. Nodes unable to compute their updates across the entire mini-batch provide partial updates, thus resulting in variable batch-size per-node. OmniLearn [274], [276] allocates batch-size to each node based on its compute capacity. Batches across nodes are balanced among each other based on PID controller (proportional-integral-derivative) [57]. It also performs weighted aggregation where updates from nodes with larger batches is given more significance than small-batch updates. Similarly for asynchronous training, OmniLearn extends its PID control mechanism to allocate worker batches that mitigates staleness issues across the whole centralized cluster. To preserve statistical performance, it also performs learning-rate scaling where local updates' step-size is scaled proportional to its training batch-size. Other techniques like speculative execution can replicate the tasks on slower nodes onto faster machines and attenuate stragglers. However, the replacement node might not be able to eliminate stragglers altogether if there is a high degree of variance in resources across nodes. Asynchronous staleness SGD [318] scales the learning-rate with respect to gradient staleness in asynchronous training and shows good convergence on image classification benchmarks. Delayed aggregation methods [305] reduce the impact of stragglers or staleness by collecting updates from training nodes at later stages instead of immediately collecting updates.

C. Data Movement and I/O

It is broadly known that data movement and I/O overhead is a performance killer in HPC and distributed computing [261], and tends to be energy-intensive as well [93]. These costs can arise due to overheads of limited

latency and bandwidth in distributed training across multi-core and multi-node systems. In the latter, communication intensity rises as we scale from data-parallel → model-parallel → tensor-parallel [257]. Data movement costs also stem from memory hierarchy, where the memory closest and fastest to the processor is often the smallest (both in CPU and GPU) [13]. Locality and data access patterns also affect the scaling of the training procedure.

Techniques to attenuate data movement or communication costs associated with different parallelization strategies are explained in §4. Remote Direct Memory Access (RDMA) enabled MPI and GPUDirect technologies [122], [113], [290], [167] further mitigates data movement cost in distributed training.

Another popular approach to minimize movement and I/O cost in distributed DL is training with a larger batch-size. For training dataset of size D, the number of iterations to complete an entire pass of the data (i.e., one epoch) with batch-size B is $(\frac{D}{B})$. Training a model to E epochs would thus require $(\frac{ED}{B})$ iterations. From this relation, it is apparent that a large batch-size would need fewer iterations to complete the same number of total epochs, leading to fewer parameter updates for the same amount of training. While it may raise the computational overhead on account of more samples to process, this approach reduces the repetitive sampling, batching and loading of training data that would be more frequent when using smaller batches. Larger-batches lead to better utilization of resources and improves the overall computational throughput. Additionally, large batch-size produces less noisy gradients which are closer to the true gradients (had we performed full gradient descent over the entire training dataset at *every* iteration). However, large-batch training also results in considerable generalization loss [135], [117] over its small-batch counterparts. The observation is that large-batches tend to converge to sharp minima, while small-batch training generally converges to a flatter minima [307], [152]. The memory usage also rises with a greater batch-size due to larger activation memory footprint over backpropagation [276]. To attenuate the generalization gap in large-batch training, prior works scale the learning-rate proportional to the batch-size; if the batch-size is raised by a certain factor, the learning-rate is increased proportionally to preserve the quality of SGD update [119]. [256] observed the optimal batch-size that resulted in best convergence to be proportional to the learning-rate and size of the training dataset. Post-local SGD [186] improved large-batch performance by synchronizing model updates initially, followed by local-SGD in later stages. Other approaches add noise into gradient updates to enhance model generalization at large-batches [297]. [185] extrapolated a smooth optimization trajectory to avoid sharp minima, such that gradients are computed over an extrapolated point which is different from the current model state. LARS and TVLARS perform layer-wise adaptive learning-rate scaling based on gradient magnitude relative to its weights [309], [308], [142]. Although these techniques improve convergence over vanilla large-batch training, they do not necessarily achieve the same model quality as small-batch training. Additionally, these large-batch methods have not yet been verified over recent LLMs.

4. Optimizing Data and Communication Cost

Over the years, size and complexity of DNNs has increased exponentially, especially in recent years with the development of large language models. A 10 Billion parameter model, a modest size by today's standards would measure 40 GB when storing parameters as 32-bit single precision floats, or 20 GB under half-precision. Training such massive models presents unique challenges, further exacerbated in distributed training scenarios which is necessary as large models expect even larger datasets to be viable.

In an ideal world, if throughput of a machine training a DNN is $\mathbb{T}_1$, then using N such machines collaboratively should increase the overall throughput to $N\mathbb{T}_1$. In real-world settings, the combined throughput of using N machines together $\mathbb{T}_N < N\mathbb{T}_1$. The deviation between the two is measure by metrics like *relative throughput* or *scaling efficiency* ($\eta_{scaling}$) which is measured as $\mathbb{T}_N/N\mathbb{T}_1$. A $\eta_{scaling}$ of 1.0 implies the ideal setting where throughput increases linearly with the number of nodes. Expensive communication costs in synchronous distributed training is the scaling bottleneck and prevents linear scaling of DL applications. Communication overheads are generally higher due to large model size to be aggregated, high-latency or low-bandwidth networks. As a consequence of Amdahl's law [30] for strong scaling, program speedup is limited by the serial portion that cannot be split across multiple processors [275]. With respect to distributed DL and $\eta_{scaling}$, communication is that serial part that cannot be parallelized. For clarity, the parallelizable part of training is processing computing loss and gradients on different mini-batches independently across different workers. At the same time, we usually refer to weak scaling when distributing a task across multiple nodes. In weak scaling, the global mini-batch size is increased as more workers are added; scaling training from 1 to N workers increases mini-batch size from b to Nb. Thus, even by

Gustafson's law [30], scaling is still limited by the serial portion of synchronous training, i.e., gradient/parameter communication.

Communication is thus the major bottleneck in distributed training [277], [320]. High-bandwidth networks may not necessarily ameliorate this slowdown as low utilization at high bandwidth might be CPU bound [320]. This is because Transmission Control Protocol (TCP) can be CPU-intensive, especially at high speeds [190]. [320] concluded its the poor implementation of network transport cannot fully utilize the available bandwidth for distributed training.

To improve the scaling efficiency and minimize communication overhead, various approaches have been proposed, including efficient communication scheduling, gradient compression and in-network aggregation on smart NiCs (network interface cards). We give a detailed explanation of these approaches in subsequent sections.

A. Communication Scheduling

Although multiple instances of worker can be launched in parallel under data-parallel training, the inner working of a DNN itself is challenging to parallelize because of the sequential, bidirectional flow of execution on a model's computational graph. Consider the case of a simplistic multi-layer perceptron (MLP) with many hidden layers. In the forward pass, computations are performed from the input to output layer to predict the output and evaluate loss, while information flows from the output to input layer in the backward pass.

Since graph execution of a DNN as well as the communication step in synchronous training is sequential by default, it is necessary to have efficient communication scheduling mechanisms that overlap computation with communication:

- *Wait-free Backpropagation (WFBP)* [316] aims to reduce synchronization overhead by overlapping computation and communication based on the structure and density of a model's computational graph. In DNNs, a major chunk of overall computation is performed in the initial and middle layers that contain fewer parameters, compared to end layers that contain most of the total parameters but only responsible for a fraction of end-to-end computation time. For example., models like VGG and AlexNet contain convolutional layers in the initial and middle layers, and fully-connected layers near the output end. Fully-connected layers account for $> 90\%$ of the total parameters of the model while performing considerably less operations. On the other hand, fewer parameters of convolutional layers account for about 90% of the total computation time. In WFBP, this property is exploited by eliminating the wait period on the completion of the entire backpropagation operation. Instead, updates for the outer layers are sent out while still computing gradients for the inner/initial layers, thus overlapping computation with communication. And so, communication-heavy dense, outer layers are transmitted while computation-heavy inner layers are still processing gradients.
  Merged gradient WFBP (MGWFBP) [248] further optimizes WFBP by combining layers with fewer parameters to transmit into one. The key insight is that merging multiple short communication tasks into a single one reduces overall communication time. Thus, layers with fewer parameters like convolutional and recurrent layers batch their parameter updates together instead of transmitting it on a per-layer basis. Compared to WFBP and synchronous SGD, MGWFBP is about $1.75\times$ and $1.45\times$ faster on a 64 node cluster.

- *TicTac* [129] optimizes computation-communication overlap in order to reduce the variance in iteration times for parameter server-based synchronous training. During downward communication from the parameter server when workers receive model state at the beginning of an iteration, they are not consumed simultaneously. Instead, parameters are consumed based on the dependency of the model's computational graph. Thus, training may accelerate or slowdown given a particular schedule of parameter transfer. TicTac estimates the ideal schedule of parameter transmission so as to minimize the slowdown on account of dependency between parameters (i.e. dependency in the graph). The metrics used to solve this NP-hard (non-deterministic polynomial) optimization problem are the partitioned graph (i.e., computational graph with a computation or communication tag associated with each op) and a timing oracle the predicts the completion time of any given op. The time oracle is implemented by tracing each op using the TensorFlow profiler. To limit the possible schedules to a valid set of orders, a priority number is assigned to each op in the partitioned graph. For a given schedule, upper and lower bounds on makespan (end-to-end job completion time) using the a oracle, such that worst makespan

is computed assuming sequential execution of one operation at a time while best makespan assumes all ops are executed concurrently. In *TIC* or *Time-Independent communication scheduling*, high priority on the partitioned graph are assigned to least blocking ops and the timing aspect (i.e., the time oracle) is not utilized. In *TAC* or *Timing-Aware communication scheduling*, higher priorities are assigned to ops that maximize computation-communication overlap and minimizes the makespan. For the latter, one needs the execution time of each op from the time oracle and op dependencies from the computational graph. Thus, TicTac can use either one of the scheduling mechanisms described here. In its evaluation, TicTac reduces straggler impact by 2.3× while improving throughput by 19% in training and up to 38% in inference tasks.

- *Priority-based Parameter Propagation (P3)* [141] uses domain-specific knowledge of a given DNN structure and overlap computation-communication in order to minimize training iteration time. In the backward pass of current iteration t, gradients are computed from output to the input layer. When data is consumed for forward pass at iteration $(t+1)$, data moves from input to final layer. Thus, there is considerable time gap between the outer layers of a DNN which should be considered when scheduling communication. Additionally, the layer-wise granularity offered by most DL training frameworks like TensorFlow and PyTorch is not necessarily optimal for parameter synchronization. P3 optimizes these issues with *parameter slicing* and *priority-based updates*. A layer is further split into smaller slices and transmitted independently in parameter slicing, and these slices are communicated based on their priority levels in priority-based updates. The priority of a slice is assigned based on when its required again in the next iteration (as described earlier with iterations t and $t+1$). Since P3 splits model layers into smaller slices/buckets, it is agnostic of a model's size and structure. It also performs well in low-bandwidth networks by optimally overlapping computation-communication and utilizing available bandwidth efficiently. Implemented on MXNet, P3 improves overall training throughput of ResNet-50 [131] and VGG-19 [254] by 25% and 66% over no-overlap synchronous training.

- *ByteScheduler* [213] is a communication scheduler that optimally partitions and reorders tensor slices/buckets before parameter synchronization. It has plugins for TensorFlow, PyTorch and MXNet, supports both parameter server and allreduce communication collectives and uses either one of TCP or RDMA for synchronization. Communication effects vary depending on model size and complexity, communication mechanism (PS or allreduce), available network bandwidth, cluster-size, etc. For e.g., in PS strategy smaller tensor slices are more efficient as pull and pull operations can utilize bi-directional network bandwidth. On the other hand, Allreduce prefers larger tensor partitions as each slice incurs synchronization overhead across all workers. Sychronization costs increase as more workers are added in the distributed training cluster as well. ByteScheduler addresses these challenges by using an auto-tuning algorithm based on Bayesian Optimization, which adaptively optimizes tensor partitions and credit sizes based on current system and model parameters. Declarative engines like TensorFlow and MXNet build a dependency graph, while imperative engines like PyTorch execute operations in a FIFO (First-in, First-out) manner. Regardless of the ML framework, a scheduler should be able to delay a communication op while keeping the original dependencies in the underlying framework engine. Due to graph dependency, it should also not schedule a communication op before the engine allows. In order to do both, ByteScheduler proposes a *dependency proxy* mechanism and *layer-wise out-of-engine* dependency. A proxy is an operation that can be inserted into a framework engine and claim dependencies to and from other operations. In PyTorch-like imperative engines, this is done via hooks, while declarative engines provide simple APIs to create such ops and define dependencies. Running ByteScheduler on a 128 GPU cluster with 100 Gbps (gigabits per second) bandwidth outperforms P3 by up to 43%, and speeds up models like AlexNet [158] and VGG19 [254] by 96% and 60% respectively over vanilla ML frameworks without any optimized communication scheduling.

- *PACE* [60] is another communication scheduler that combines domain knowledge of ML execution engines and default tensor sizes for transmission. Other schedulers assume a layer-wise computation graph while framework engines (e.g., TensorFlow) assume a directed acyclic graph (DAG) as the execution model. Additionally, default tensor sizes in DNNs have high variance; some tensors are very small while some are large. Thus, transmitting one tensor at a time may not be bandwidth-optimal or may cause network saturation, implying inefficient communication. PACE preemptively schedules and fuses tensors via allreduce collective based on

the DAG generated by the underlying engine, which maximizes both bandwidth utilization and overlap between computation-communication. It does so by launching a DNN training job for only a few iterations and collecting the model DAG, time taken by each op in the DAG and tensor sizes of all tensors using the allreduce op. It then updates the DAG by combining smaller tensors into a combined allreduce call which generates a communication schedule of allreduce ops via an optimization solution that minimizes completion time of the latest computation operators and the DAG's execution time. With these optimizations, PACE outperforms Horovod [242] and WFBP [316] by around 30%, and TicTac [129] by achieving 17% more speedup while training VGG16 [254].

B. Compression in Deep Learning

Compression is employed to reduce the size or volume of a particular target. In the context of deep learning, compression is generally applied across one of the three modalities: to compress *training data*, compress the DL *model* itself, or compress *gradients* or *model updates* in distributed training. Data compression techniques help reduce the quality or quantity of the data used for DNN training. Model compression decomposes the size of the DNN itself, either to reduce its computational overhead or its size for faster inference on edge/mobile devices (i.e., for model serving). In distributed training, gradient compression is used to reduce the volume of updates to be synchronized among multiple nodes, thereby lowering the overall communication overhead. We describe these techniques as follows:

1) Data Compression

- *ZFP* [188] is an error-bound, lossy compression library designed to support high compression ratios on floating-point and integer arrays, while additionally supporting a lossless compression version. Large batches of data samples such as images can thus be compressed to a smaller footprint to train models with smaller inputs, while simultaneously lowering the memory requirements. ZFP supports various backends such as OpenMP, CUDA [2] and HiP [21].
- *SZ* [96] is a lossy compression technique which reduces the size of scientific datasets (such as those used for climate modeling, fluid dynamics, turbulence flow, molecular simulations etc.) at the cost of some degree of precision. It provides a user-specified error tolerance threshold, so datasets with distinct characteristics can have varying degree of compression loss.
- *SZ2* [183] attains higher compression ratios and improves SZ algorithm using a prediction scheme that evaluate a data point based on its neighbors and encodes the residual between the actual and predicted values.
- *SZ3* [321] improves the compression-decompression overhead of SZ2 using multiple prediction strategies and error layers for high accuracy across multidimensional and numerical datasets. Its additionally handles different precisions such as FP32 and FP64.

2) Model Compression

One of the most common and popular techniques for compressing the DNN model size is *pruning*, which refers to the technique of removing redundant or less significant model parameters during the course of training itself, or during the inference stage once the model is trained. The use case for the former is to reduce the model size during synchronization step, while the latter is done to reduce inference time (i.e., only forward pass) and minimize memory footprint of a given model [198]. The optimal pruning technique remove weights that are sensitive to model performance and do not degrade model generalization significantly. Additionally, pruning techniques work equally well on both untrained and pretrained models. In prior art, *dropout* [260] acts as a pruning mechanism in DNNs so as to counter the over-fitting problem in deep learning. In dropout regularization,some neurons are randomly dropped out during training based on a chosen dropout factor. Although the actual size of the model is not reduced, some connections are skipped which reduces the amount of computation that needs to be performed in an iteration.

- *Optimal Brain Damage (OBD)* [165] removes insignificant weights from a neural network by using second-derivative information from the Hessian matrix to make trade-offs between a model's network complexity and training error. OBD defines the saliency of a parameter as the change in the objective/loss function if the said parameter was removed. For optimality,the saliency can be predicted analytically by constructing a local model

of the loss function as a Taylor series and using second-order information for a specific parameter. In this technique, we train a model for a certain duration, and compute the Hessian matrix for the second-derivative as well as the saliencies of each parameter. Low saliency parameters are deleted from the computed saliencies and the process is repeated iteratively.

- [259] demonstrates how similar neurons in a DNN are redundant by removing one neuron at a time instead of removing individual weights. The key insight is that when two weight sets are the same, one of them can effectively be removed. Consider a model with two parameters (w_1, w_2), input X, and non-linear function $h(\cdot)$, then $z = a_1 h(w_1^T X) + a_2 h(w_2^T X)$. If $w_1 = w_2$, then $z = (a_1 + a_2)h(w_1^T X) + 0 \cdot h(w_2^T X)$. Weight w_2 can be removed since its the same as w_1 and already added to the first term. Neurons which are equal (i.e., fire together) can be wired together with via a *surgery step* (by changing a_1 to $a_1 + a_2$ in the given example). Equal weights thus contribute to redundancies in DNNs and can be removed without any considerable degradation in final model accuracy. This work extends on parameter saliency [165] by computing saliency on all possible pairs of weight-sets and picking the minimum values and deleting the corresponding neurons. The authors achieve same performance as baseline AlexNet [158] even after removing 35% of the total model parameters.

- [126] optimizes the computation cost and model size by pruning redundant connections without sacrificing the final accuracy for model inference. It first trains the network to learn significant weights and connections, followed by removing the unimportant ones. To hone the updated network, the model is trained again to fine-tune remaining parameters. Although the training time is not reduced in this case (it increases further!), models like AlexNet [158] are compressed by $9\times$ while attaining the same accuracy as uncompressed model on ImageNet dataset [94], while larger models like VGG19 were compressed by a factor of $13\times$.

- *HashedNets* [82] is a specialized network architecture that compresses DNNs by using a hashing function that randomly groups parameters into hash buckets. All parameters within a given hash bucket share a common value. With more training, the shared parameters are fine-tuned in HashedNets to achieve optimal values.

- *Sparsity-informed Adaptive Pruning (SAP)* [97] defines PQ Index (PQI) to measure sparsity (i.e., potential compression) that can be applied to DNNs without sacrificing the accuracy. The key property of PQI is that first decreases when is model is effectively regularized and increases when the degree of applied compression reaches the point corresponding to the beginning of underfitting. On the other hand, PQI decreases on applying too much compression, accompanied with degradation in model generalization. For a given set of compression hyperparameters, $0 < p < q$, PQI is defined as $1 - d^{\frac{1}{q}-\frac{1}{p}}||w||_p/||w||_q$, such that $||w||_p$ is the $\mathbb{L}_p$-norm of weights w_p. Beginning with a mask of ones on the model parameters, a DNN is trained for a certain duration. Based on the computed PQI, the lower bound on the number of retained parameters and total pruned parameters are computed. A new mask is then generated based on removing the least magnitude weights corresponding to the total pruned parameters to remove. This process is repeated and model parameters are thus trimmed off over the iterations.

3) Gradient Compression

Orthogonal to building better and more efficient communication scheduling mechanisms, compression has been a promising avenue to reduce the communication cost of distributed synchronous training [304], [265]. In addition to training, compression is also critical for deployment of large DNNs for inference tasks on memory-constrained edge/mobile devices. DL models initially used single-precision or 32-bit floating point representation for both model weights and gradients. For e.g., ResNet152 [131] translates to 60 megabytes of floats, VGG19 [254] occupies 144 MB while a vision transformer [99] takes 330 MB of space. In the recent LLM era, mixed-precision and 8-bit float have become the training norm [104].

Large language models like GPT-3.5 [38] would consume about 652 GB of memory space just to hold the model parameters (in single-precision or FP32 representation). Storing intermediate data and activations, as well batches of training data that need to be loaded into memory requires additional space. Thus, communicating model updates at every iteration can be very expensive, be it training large language models in high-speed data-centers and cloud settings, or training models like ResNet and VGG across mobile devices in a federated setting with geographically

distributed nodes and networks with limited bandwidth. Limited memory on specialized ML accelerators like GPUs and TPUs is another reason why it is critical to develop optimal compression mechanisms. To facilitate research and development in optimal compression-based deep learning, generic frameworks like GRACE [303] have also been developed that come with different compression mechanisms right out of the box and support multiple DL engines like TensorFlow and PyTorch.

Depending on the compression technique and the degree of compression desired, a compression technique may have a certain overhead that makes it viable or impractical for DL-based compression [51]. In typical synchronous training, iteration time $t_{iter} = t_{compute} + t_{sync}$, where $t_{compute}$ is time taken to compute loss and gradients and t_{sync} is the synchronization time to exchange parameter updates of the entire model. With compression, the total iteration time $t_{iter} = t_{compute} + t_{compress} + t_{sync}^{(z)}$, where $t_{compress}$ is the compression overhead and $t_{sync}^{(z)}$ is the communication time for gradients compressed to a degree of z. A common technique used to achieve high compression with a variety of compression methods is via *residual gradient accumulation* or *error-feedback* mechanism [52], [151], [327]. In error-feedback, gradients on worker n target a Compression Factor (CF) of z via operator $\mathcal{C}(\cdot)$ at iteration t where gradients $g_{n,t} = \frac{\partial \mathcal{L}}{\partial w_{n,t}}(x_{(n,t)}, w_{n,t}) + r_{n,t-1}$ are compressed to $c_{n,t} = \mathcal{C}(g_{n,t}, z)$. The residual gradients are the gradients that are not eligible for compression (as determined by $\mathcal{C}(\cdot)$), and can be computed for the next iteration by the symmetric difference between the original and compressed gradients in the current iteration: $r_{n,t} = g_{n,t} - c_{n,t}$. Notice how the gradients $g_{n,t}$ at iteration t are appended with residual gradients $r_{n,t-1}$ from iteration $(t-1)$. By doing so, the progress made by workers that does not make it in compression is not discarded, but accumulated over the iterations. Once the accumulated gradients corresponding to a parameter are significant enough to meet the compression criteria, they are communicated and the associated residual gradient entry becomes zero. It is important to note that the size of residual gradients is the same as the original gradient size. Once the compressed gradients are transmitted, parameters are tweaked as per SGD update: $w_{n,t+1} = w_t - \eta \frac{1}{N} \sum_{i=1}^{N} c(n,t)$.

Gradient compression techniques can be broadly classified in the context of deep learning as follows: *sparsification*, *quantization* or *low-rank approximations*.

a) Sparsification:

Compression methods based on sparsification select a subset of values from the original tensor and setting the remainder to zero [263], [55]. In the context of deep of distributed deep learning, the gradients computed by each worker can be sparsified by selecting a fraction of the values and assuming the remaining to be zero based on some rule or heuristic. Gradients can be compressed by an operator $\mathcal{C}(\cdot)$ that selects $p\%$ of the total values in the tensor; gradients are then said to be sparsified to a *compression ratio (CR)* of $p\%$ or a *CF* of $\frac{1}{p}\%$. For e.g., setting p to 10% means a CR of 0.1 or a CF of $10\times$. Based on the degree of applied sparsification, sparse tensor thus have a considerably smaller size than the original gradients. Gradient sparsification may be static or dynamic; the tensor size remains fixed in the former while the latter falls under the category of adaptive gradient methods as the degree of sparsification varies through the course of training. Another important consideration with gradient sparsification is that in addition to the gradient values, individual indices corresponding to the values in those tensors needs to be transmitted as well. If the tensors to be transmitted contain M cumulative gradient,compressing to $p\%$ returns $\frac{Mp}{100}$ values. When synchronizing updates in distributed training, $\frac{Mp}{100}$ values have to be transmitted in conjunction with $\frac{Mp}{100}$ indices; thus $2\frac{Mp}{100}$ data points are communicated in total. Since the indices of the values chosen in the compressed gradient vary across workers, using bandwidth-optimal algorithms like ring-allreduce is not possible in gradient sparsification [51]. In centralized systems like parameter servers, the indices and values have to be two separate *push* calls from each worker to the PS, while decentralized systems using MPI-based collectives use *all-gather* to synchronize their updates.

- *Top-k* compression [52], [55], [263], [249], [247] is a sparsification method that select the top $k\%$ most significant values from the gradient tensors, and setting the remainder to zero. The significance of a parameter's gradient is measured by considering only the magnitude and ignoring the sign associated with the gradient. The top $k\%$ values can be computed using a max heap and sorting the top $k\%$ values out of M gradients in $\mathcal{O}(M + k \cdot logk)$ time, although other efficient implementations exist as well. Communication overhead is thus considerably reduced by having to transmit sparse instead of dense updates. As with any sparsification method, the values and indices from each worker have to be transmitted separately. [52] used top-k compression with error-feedback to sparsify 99% of the gradient updates while training on the MNIST dataset [164] and achieved

49% speedup. [263] and [55] provides theoretical convergence guarantees for$k$-based sparsification techniques (like top-k and random-k) and show how this sparsification-based compression technique achieves the same convergence rate as vanilla SGD when equipped with the error-feedback mechanism.

- *Gaussian-k* [247] studies the distribution of gradients to propose an approximation of the Top-k compression algorithm. Compared to the computational overhead of Top-k that uses sorting to find optimal gradients, Gaussian-k is more efficient on both CPUs and GPUs. Thus, the latter has a better scaling efficiency than naive Top-k compression. The mean and standard deviation is computed at each iteration for gradients with a near-Gaussian distribution, which is used to estimate the threshold with a quantile or percent point function (ppf). Since the gradients have a near-normal and not an exact normal distribution, the estimated threshold could be skewed. Thus, threshold is moved up and below a certain degree of the estimated value to attain accurate top-k values.

- *Deep Gradient Compression (DGC)* [187] interprets residual gradient accumulation as increasing the batch size such that it increases by a factor corresponding to the update interval when a particular gradient value becomes eligible for communication. DGC further optimizes sparsification for momentum SGD (Equation (9a)) by accumulating the residual velocity instead of the actual gradients and applying compression with operator $\mathcal{C}(\cdot)$, shown by Equation (9b).

$$w_{t+1} = w_t - \eta \frac{1}{N} \sum_{i=1}^{N} u_{i,t} \text{ where velocity } u_{n,t} = m \cdot u_{n,t-1} + g_{n,t} \text{ and } g_{n,t} = \frac{1}{|b|} \sum_{x_{(n,t)} \in \mathcal{B}_n} \frac{\partial}{\partial w_{n,t}} \mathcal{L}(x_{(n,t)}, w_{n,t}) \tag{9a}$$

$$u_{n,t} = m \cdot u_{n,t-1} + g_{n,t} \text{ with residual velocity } v_{n,t} = v_{n,t-1} + u_{n,t} \text{ and } w_{t+1} = w_t - \eta \frac{1}{N} \sum_{i=1}^{N} \mathcal{C}(v_{i,t}) \tag{9b}$$

As gradients are accumulated over the iterations, training could possibly encounter the exploding gradient problem [63]. To counter this issue, DGC applies local gradient clipping to the computed gradients before they are accumulated with the residual gradients. Due to delayed update of low-value gradients due to their gradual accumulation, staleness may creep in over the course of training. To mitigate staleness effects, DGC incorporates *momentum factor masking* and *warm-up training*. As described in §3-A0b, staleness stems from asynchronicity which in turn can be attributed to implicit momentum [197]. DGC applies the same momentum mask to accumulated/residual gradients $v_{n,t}$ and momentum factor $u_{n,t}$, which inhibits momentum for delayed gradients, minimizing the effects of staleness. In warm-up training, instead of directly compressing gradients to CF z from the first training iteration itself, we start training without any compression and gradually increase the CF from $1\times$ to z over a certain duration of steps/epochs. For models like ResNet-50 [131] and DeepSpeech [127], DGC achieved CFs from $270\times$ to $600\times$ without losing any accuracy.

- *Random-k* [263] sparsification works by randomly choosing a subset of values from the accumulated gradients. The number of gradients selected for communication is determined by chosen CF of k. Due to its randomized selection procedure, this method has negligible compression overhead compared to other sparsification techniques. Thus, iteration time is mostly determined by the computation time and the time taken to communicate sparse gradients. However, Random-k does not scale well to high CFs; models can converge with $10\times$ CF in some cases, but tend to diverge beyond that, causing severe degradation in a model's train and test accuracy [46].

- *Sparsity-Inducing Distribution-based Compression (SIDCo)* [46] works by modeling residual gradients at every iteration as a sparsity-inducing distribution (SID) and sparsify the gradients using a threshold calculated from the SID. The intuition is that large DNNs tend to be over-parameterized and as training progresses, a model eventually starts converging. Thus, most of the gradients are zero or close to zero. SIDCo thus exploits this inherent sparsity in DNN training by using distributions like double Exponential, double Gamma and double Generalized Pareto (GP). Since its an approximate approach for threshold-estimation, it can be inaccurate at higher CFs. To solve this problem, SIDCo uses a multi-stage threshold estimator where the original gradients

are fitted to a SID with a CF z_1. Sparse gradients generated from before are then used to fit another SID to compress gradients to CF z_2. The sparse gradients returned thus have a CF of $z_1 \cdot z_2$ with respect to the original gradients and are much more accurate than directly compressing accumulated gradients to CF $z_1 \cdot z_2$ with SIDCo. Compared to techniques like naive Top-*k* and DGC, SIDCo has a considerably lower compression overhead. In its evaluation on various convolutional and recurrent networks, SIDCo accelerates training by up to 42×.

- *Threshold-v* [52] is a sparsification technique where a fixed hyperparameter *v* is assigned prior training. Based on the value of *v*, compression is performed by selecting gradients with magnitude greater than or equal to *v*. The compression overhead of Threshold-*v* is minimal as only a mask vector is applied to select values $\geq |v|$.

- *gTopKAllReduce* or *global Top-k AllReduce* [249] is an allreduce-friendly gradient sparsification technique. In naive Top-*k* where gradient values and indices have to be synchronized (i.e., a total of $2k$ values) via all-gather takes time $\log(N)\alpha + 2(N-1)k\beta$ on N workers on a network with latency α and bandwidth $1/\beta$. As the second term increases with the number of workers, communication becomes expensive on larger cluster sizes. Thus, this approach is not optimal for low-bandwidth networks. gTopKAllReduce designs a technique for top-*k* compression with a communication mechanism that is bandwidth optimal. In the mechanism, communication is initiated between a pair of nodes at every step via *send* and *recv* collectives. With a model comprised of M differentiable parameters, a worker sends $2Mk$ values (each Mk corresponding to eligible gradients and indices) to another worker. The worker that receives the values aggregates the result and selects top-*k* values. By the end of first step, there are $N/2$ workers with top-*k* gradients. These steps are repeated $\log_2(P)$ times. For e.g., a cluster of 8 workers involves $\log_2(8)$ or 3 rounds of *send* and *recv* calls between pairs of workers. By the end of $\log_2(P)$ rounds, a single worker will have the *global top-k* values that are distributed to all workers via *broadcast*. The overall time complexity of global top-*k* is $2\alpha\log(P) + 4Mk\beta\log(P)$. Unlike all-gather, the time in global top-*k* does not increase linearly with the number of workers and is thus bandwidth-optimal.

b) Quantization:

In this class of compression, updates from workers are quantized either by reducing the bit-width of single-precision floats or via a codebook such that each gradient value and index corresponds to pre-defined state, effectively bounding or quantizing the range of values that can be taken by the compressed tensors [54]. In the former, one can reduce 32-bits to a minimum of 1-bit, thus achieving a maximum possible compression of 32×. Similar to sparsification, quantization techniques benefit greatly from error-feedback and residual gradients as well.

- *1-bit SGD* [241] aggressively quantizes 32-bit gradients to 1-bit, thereby achieving a compression of 32×. To prevent accuracy deterioration, 1-bit SGD adds the quantization error back to the gradients computed in the subsequent iteration (i.e., error-feedback). For aggregation of quantized gradients, each trainer among N workers receives $1/N$-th of the quantized updates, decompressed, reduced and quantized again before being re-distributed to all workers which is then applied in the SGD update.

- *Threshold-v* quantization [264] also improves convergence via error-feedback. Unlike Threshold-*v* sparsification, this method quantizes gradients by choosing a fixed threshold *v* and encodes all values greater than +*v* and less than $-v$ with 1, while setting every other value to 0. That is, gradients with magnitude greater than or equal to *v* are communicated while everything else is discarded.

- *Quantized SGD (QSGD)* [54] offers flexible compression by trading between communication bandwidth and convergence time. QSGD allows workers to trade-off the number of quantized bits exchanged per-iteration with additional variance. It uses stochastic quantization such that gradients are quantized by randomized rounding while still keeping intact the statistical properties of the original gradients. QSGD then generates encodings with an efficient lossless code for quantized values. For a gradient tensor g, element i is compressed by operator $\mathcal{Q}(g_i) = ||g||_2 \cdot \text{sign}(g_i) \cdot \epsilon_i(g, s)$ where $\text{sign}(g_i) \in \{-1, +1\}$. Here, random variables are chosen by function $\epsilon(\cdot)$ (in Equation (10)). With user-specified integers $0 \leq l < s$, $|g_i|/||g||_2$ are quantized to $[l/s, (l+1)/s]$.

$$\epsilon_i(g,s) = \begin{cases} l/s & \text{with probability } 1 - (\frac{|g_i|}{||g||_2} \cdot s - l) \\ (l+1)/s & \text{with probability } \frac{|g_i|}{||g||_2} \cdot s - l \end{cases} \tag{10}$$

- *TernGrad* [296] quantizes gradients across three levels: $\{-1, 0, +1\}$. Using residual gradients from error-feedback, TernGrad develops a layer-wise quantizing approach with gradient clipping to improve convergence. A tenary operator $\mathcal{T}(\cdot)$ quantizes element g_i from the accumulated gradients g as $\mathcal{T}(g_i) = \max(\text{abs}(g_i)) \cdot \text{sign}(g_i) \cdot b_i$. The first term extracts the largest magnitude to use for the quantized gradients, the second term collects the sign of the corresponding gradient element, i.e., $\text{sign}(g_i) \in \{-1, +1\}$ and the last term follows a Bernoulli distribution and takes 0/1 based on Equation (11).

$$b_i = \begin{cases} 1 & \text{with probability } \frac{|g_i|}{\max(\text{abs}(g_i))} \\ 0 & \text{with probability } 1 - \frac{|g_i|}{\max(\text{abs}(g_i))} \end{cases} \tag{11}$$

- *Variance-based quantization* [273] builds on the observation that most gradient updates can only be accumulated (and not communicated) until a high magnitude, low-variance update is eventually collected in the residuals. An update is considered insignificant and can thus be ignored when its value is small over its variance across the accumulated iterations. The residual gradients are effectively perceived as delayed updates. The condition for a gradient value to be eligible for compression is given by Equation (12), and compares the squared mean of gradients (right side of Equation (12)) and sum of squared gradients (left side of Equation (12)).

$$\alpha \sum_{x_{n,t} \in \mathcal{B}_n} (\frac{1}{|b|} \frac{\partial}{\partial w_{n,t}} \mathcal{L}(x_{n,t}, w_{n,t}))^2 \quad < \quad (\sum_{x_{n,t} \in \mathcal{B}_n} \frac{1}{|b|} \frac{\partial}{\partial w_{n,t}} \mathcal{L}(x_{n,t}, w_{n,t}))^2 \tag{12}$$

  The parameter α controls the estimation accuracy by checking if the accumulated gradients still decrease the loss function. The gradients that satisfy the above condition are then compressed via quantization and encoding the corresponding indices. A CF of $8\times$ is attained by quantizing 32-bit gradients to 4-bits.

- *signSGD* [64] applies quantization for reducing communication costs by transmitting only the sign of the gradients computed in a given iteration. Given an element i in gradient g, we thus have $\text{sign}(g_i) \in \{-1, +1\}$. The quantized gradients are aggregated based on the majority vote where each worker sends its sign associated with a particular gradient. A modified version of signSGD called *SIGNUM* [65] applies momentum to the equation and transmits the sign value of that for optimized communication. *EF-signSGD* [151] adds error-feedback to signSGD and has much better generalization than just using the latter, and achieves the same convergence rate as vanilla SGD. In addition to using the residual gradients, EF-signSGD scales the magnitude of the gradient updates as well. The gradients g_t at iteration t are updated with accumulated gradients $r_t = g_t + r_{t-1}$, which are then quantized by EF-signSGD compressor $\mathcal{C}(r_t) = |r_t|_1 \cdot \text{sign}(r_t)/\text{len}(r_t)$ and EF-gradients for the next iteration $r_{t+1} = r_t - \mathcal{C}(r_t)$ Model parameters are then updated as $w_{t+1} = w_t - \eta \sum_{i=1}^{N} \mathcal{C}(r_t)$ and pushed back to the workers to proceed with the next iteration.

- *Blockwise Error-Feedback SGD* or *EF-blockSGD* [327] optimizes EF-SGD which quantizies gradients based on the magnitude and sign of individual gradient values. The magnitude term of the residual gradients used in EF-SGD is $r_t/\text{len}(r_t)$ which can become very small when the model size is large and the tensors are sparse. This is a variation of diminishing gradient problem in the context of compression-based DNN training. To mitigate this, EF-blockSGD proposes to bucket gradients into B blocks, and then apply quantization on each block corresponding to its scaling factor. If the original accumulated gradients r_t computed at iteration t are split into B blocks, i.e., $r_t = r_{(1,t)} \cup r_{(2,t)} \cup r_{(3,t)} \ldots \cup r_{(B,t)}$. Quantization is then applies on each block such that scaling magnitude used for block 1 is $|r_{(1,t)}|_1/\text{len}(r_{(1,t)})$, $|r_{(2,t)}|_1/\text{len}(r_{(2,t)})$ for block 2, and so on.

c) Low-Rank Approximations:

Low-rank approximations work by decomposing gradients across lower dimensionalities. Considering the gradients as a matrix of dimensions $(p \times q)$, i.e., $G \in \mathbb{R}^{p \times q}$, it can be decomposed into lower-rank matrices U and V such that $U \in \mathbb{R}^{p \times r}$ and $V \in \mathbb{R}^{r \times q}$. This is based on the assumption that DNNs are over-parameterized and tend

to have a low-rank structure. Instead of communicating G, we instead transmit U and V, achieving compression as fewer gradient values are synchronized in this manner. The CF attained here is $p \cdot q/(p \cdot r + r \cdot q)$.

- *PowerSGD* [286] is an allreduce-friendly low-rank compression technique. Gradients for a model are decomposed to a target rank r on a per-layer basis. Fully-connected and convolutional layers and their corresponding gradients can be represented as a matrix, say of dimension $\mathbb{R}^{p \times q}$. PowerSGD then uses power iteration to decompose the gradient matrix G into two lower r-rank matrices P and Q. Here, *iteration* does not refer to the training iterations, but the times we run the power iteration algorithm in a loop to get accurate low-rank representation of G. Additionally, for the first iteration of PowerSGD, Q is initialized in an IID manner from a normal distribution. For the specified iterations, PowerSGD then computes $P = M \times Q$ and aggregates it across all workers via allreduce. Next, the matrix is normalized $\widehat{P} = P/\text{det}(P)$ (where $\text{det}(P)$ is the determinant of the matrix), followed by computing $Q = G^T \times \widehat{P}$ and synchronizing it among workers. After running for user-specified iterations, we get the compressed representation of G as $(\widehat{P}, Q)$.

- *GradZip* [285] also performs matrix decomposition and allreduce communication of low-rank updates. It decomposes gradient matrix G into r-rank matrices P and Q (like PowerSGD) to minimize the criterion that contains an additional regularizer term:

$$\max_{P,Q} ||G - P \times Q||_F^2 \; + \; \lambda(||P||_F^2 + ||Q||_F^2)$$

- *ATOMO* [291] returns a low-rank representation of the gradients corresponding to a sparsity budget s while minimizing variance by performing sparisification on quantized gradients. ATOMO performs compression by atomically decomposing original gradients as $G = \sum_{i=1}^{d} \lambda_i a_i$ where $\mathcal{A} = \{a_i\}_{i=1}^{d}$. Sparse gradients with low variance are calculated as $\widehat{G} = \sum_{i=1}^{d} \lambda_i t_i a_i / p_i$ where $0 < p_i \leq 1$ and t_i is a Bernoulli distribution of p_i.

d) Hybrid Gradient Compression:

Hybrid compression techniques techniques combine two or more compression techniques to attain model with accurate, low loss targets while still achieving high compression during training or inference. Different classes of sparsification, quantization or low-rank updates can be combined together. Under hybrid compression, It is common to combine a compressor that provides strong convergence guarantees but offers low compression with another lossy compression method that might have comparatively poor convergence but offer high degree of compression.

- *Deep Compression* [125] is a multi-stage compression pipeline for inference tasks that collectively reduced model size by up to 50× without affecting the convergence quality of DNNs. It involves parameter pruning, quantization and Huffman coding [282], such that insignificant weights are removed in pruning, then a codebook-based quantization technique is applied to share a finite set of weights across the entire model, and finally applies Huffman coding to encode model weights and indices for further space optimization. In Deep Compression, model parameters below a certain threshold and re-train the network with the smaller model, the results of which are stored in compressed sparse row or column (CSR or CSC) format. For weight-sharing based quantization, a model with M total parameters can be quantized by k clusters where each centroid is a 32-bit floating point and there are $M \;\; \log_2(k)$ indices, giving an effective compression rate $z = 32M/(M \log(k) + 32k)$. The centroids for weight sharing are computed so as to minimize sum of squares within each cluster: $\arg\min_C \sum_{i=1}^{k} \sum_{w \in c_i} |w - c_i|^2$.

- *Reduction for Synchronization Bandwidth (RedSync)* [107] is a hybrid compression technique that combines sparsification with quantization. It compresses DNNs on a per-layer basis and accumulating residual gradients over the course of training. RedSync then sparsifies the residual gradients to its target CF by using a mask vector (of 0s and 1s) and applying it element-wise on the accumulated gradients. For top-k sparsification, it proposes *trimmed top-k selection* and *threshold binary search selection*. In the former, gradients are modeled as a normal distribution and size of sparse gradients is first measured with a large threshold value (calculated using the mean and maximum value of gradients). If the size of returned tensors is smaller than than desired size (based on k), threshold is decreased by a certain fraction and re-evaluated until the final tensor size is more than k. In threshold binary selection, instead of searching for top-k elements, a threshold is evaluated for the range

top-*k* to top-*2k*. The elements larger than the calculated threshold are then selection for communication. After sparsification, RedSync then applies *Alternating Signs Quantization (ASQ)* to further compress the sparsified gradients with an additional $2\times$ compression. ASQ works by alternately quantizing between top 0.1% elements and minimum 0.1% elements in adjacent training iterations. As the top and bottom 0.1% have the same sign, no communication is required to transmit the extra sign information.

- *SparCML* [228] is a compression-friendly communication library that builds additional features on top of MPI like low-precision data communication and asynchronous operations. To reduce synchronization costs, it combines sparsification with quantization. SparCML applies top-*k* compression on the gradients appended with the local residual gradients, updates the residual gradients with gardient values that are ineligible for communication. It then calls allreduce on the sparsified gradients to aggregate updates in a sparsity-aware manner. If the reduction of compressed gradients becomes dense, the library will quantize the gradients so as to reduce the synchronization overhead. Gradient quantization is applied with QSGD [54]. When none of the zero elements in compressed gradients across workers overlap, allreduce is implemented by calling reduce on one worker, followed by a broadcast call. When all the non-zero values across all workers' compressed updates overlap, an efficient allreduce call (like ring or tree allreduce) is made.

- *Sparse Ternary Compression (STC)* [239] is a compression framework designed to reduce the communication cost in federated learning by combining upstream top-*k* compression, downstream compression, gradient ternarization and optimal Golomb encoding of weight updates.

$$\triangle \widehat{w}_{n,t+1} = \text{Top}_{k\%}(\triangle w_{n,t+1} + r_{n,t}) \;\text{ and }\; \triangle w_{*,t+1} = \frac{1}{N}\sum_{i=1}^{N} \widehat{w}_{i,t+1} \tag{13a}$$

$$r_{n,t+1} = r_{n,t} + \triangle w_{n,t+1} - \triangle \widehat{w}_{n,t+1} \tag{13b}$$

  During upstream compression, residual parameters are appended to local parameters before applying top-*k* compression, followed by averaging the sparse parameters on the central server (Equation (13a)). The updates that are *not* sent to the server are accumulated to the residual gradient for the next iteration, shown by Equation (13b). A similar compression scheme is applied downstream, i.e., from the central parameter server to the participating devices. To avoid updates from devices using stale updates from not having participating in communication for a while, such devices have to first download latest parameters from the server to be eligible for communication in the next round. The compressed top-*k*% parameters are further compressed with a ternary tensor in the range $\{-\mu, 0, \mu\}$, thus quantizing 32-bit single-precision floating points to 1-bit. To avoid sending the index positions of sparse tensors, STC uses Golomb encoding to communicate the relative distance between non-zero elements, thereby reducing the average number of position bits. STC outperforms FedAvg in terms of final model accuracy in settings with non-I.I.D. data or low participation rate in a communication step when the number of clients is large, or in small-batch training. In IID data partitioning, STC is pareto-superior to Federated Averaging [194] in that it achieves the same target accuracy in fewer iterations and with a lower communication budget.

e) Adaptive Compression Techniques

Unlike traditional HPC jobs, distributed DNN training is different in the sense that not every iteration holds the same degree of significance [47]. This is due to the stochastic nature of SGD and a model's training where gradients are large initially, but as the model converges over the iterations, gradients become sparse and smaller in magnitude. Some updates are thus more important than others during training. Hence, it is viable to develop adaptive compression methods as well that tweak CF or the compression technique applied itself as a model trains over the iterations.

- [101] builds atop threshold-*v* based quantization to develop an adaptive compression strategy. Unlike conventional threshold-*v* that uses a fixed threshold *v* for quantization, the adaptive mechanism takes a user-specified CF z and computes a positive (τ^+) and a negative threshold (τ^-) on accumulated gradients of size M at every iteration. The upper threshold τ^+ is computed by taking the top M/z updates, while the τ^- is calculated

taking the smallest M/z updates from the gradients. An encoding mask of 1s is used for gradient values meeting the upper-lower thresholds, a 0 mask is used otherwise. For bandwidth-optimal communication, a quantization-aware variant of allreduce is also implemented that first performs a reduce-scatter between a pair of workers, followed by a ring-based allgather.

- *Adaptive Residual Gradient Compression (AdaComp)* [80] keeps track of the local gradient residues via error-feedback and adaptively tunes compression rate based on the significance of the collected gradient updates. In addition to tuning CF across iterations, AdaComp can tune its CF across DNN layers as well. The accumulated gradients are bucketed into several bins of fixed length, and the maximum absolute value for each bin is evaluated. A gradient is selected for communication if the updated residual gradient at current iteration multiplied by a certain scaling factor is greater than maximum value of the bin, i.e., $m_{(b,t)} = \max(\text{abs}(r_{(b,t-1)}))$ is the largest absolute value from bin b at iteration t. Once gradients are computed, residuals are updated as $r_{(b,t)} = r_{(b,t-1)} + g_{(b,t)}$ and thus, we transmit gradients if the following condition is satisfied $m_{(b,t)} < s \cdot r_{(b,t)}$ for scaling factor s. The compressed residual gradients are quantized on a per-layer basis to further improve the CF.

- *Accordion* [50] implements an adaptive compression strategy that can be integrated with other sparsification, quantization or low-rank methods. The intuition behind Accordion is that DNN training is a stochastic process and not every iteration is equally important. Some updates are more critical than others with regard to the updates made to the model. This is corroborated by works that explore critical periods in deep learning training and training sensitivity in the early training stages [47], [111]. Accordion detects critical updates from tracking changes in gradient variance between consecutive training iterations. It switches between a low CF z_{low} and a high CF z_{high} depending on whether the gradients are important enough or not. When updates are critical, Accordion uses z_{low} for compression, and z_{high} otherwise. Formally, critical updates are detected as $(||g_{(n,t-1)}||^2 - ||g_{(n,t)}||^2)/||g_{(n,t-1)}||^2 \geq \epsilon$. Here, ϵ is a user-specified detection threshold that makes a trade-off between compression/communication time and accuracy. A low ϵ means even a slight change in the inter-iteration gradients is recognized as a critical update and thus, we use low CF z_{low} for compression which implies more communication cost. Since gradients at low CF are more representative of the original gradients, accuracy loss for the same iterations is not that severe (compared to training *without* compression). On the other hand, a high ϵ suggests that inter-iteration gradients have to differ significantly in order to be qualify as critical updates. Thus, we mostly use high CF z_{high} which brings down synchronization time. At high CF, more information is lost and thus, quality of updates is poor in this case. Thus, the accuracy attained in the same iterations at high CFs and at no-compression may be more severe.

- *GraVAC* [280] proposes an adaptive compression technique that adjusts CF on a per-iteration basis; starting with low compression initially and gradually increasing CF as training progresses. It combines the pareto-related parallel and statistical efficiency in distributed training with a single metric 'compression throughput', combining overall system throughput with 'compression gain': $T_{compression} = T_{system} \times$ *Compression gain*. The latter term accounts for statistical inefficiency associated with lossy gradient compression, and measured by the ratio of $\mathbb{L}$-2 norm of prior and post-compression gradients. If the error-fed updates are obtained by adding the backpropagated and residual gradients, i.e., $g_t^{(ef)} = g_t^{(o)} + r_{(t-1)}$, and compressed updates are generated by compressor $\mathcal{C}(\cdot)$ such that $g_t^{(c)} = \mathcal{C}[g_t^{(ef)}]$. Compression gain is then measured as $\frac{\mathbb{E}[||g_t^{(c)}||^2]}{\mathbb{E}[||g_t^{(ef)}||^2]}$. Using these metrics, GraVAC develops an adaptive strategy where models use low CF in early and critical training phases, and higher CF as the model begins to saturate and gradients get smaller. Compression can be increased via different CF scaling policies, and based on a pre-specified gain threshold, the communicated updates may be compressed to currently used CF, or the minimum CF specified in GraVAC, or the original gradients $g_t^{(o)}$ if the former two trim excessive information and fail to update the model in any significant way.

- *All-Reducible Top-k* or *AR-Topk* [279] is an adaptive compression-communication mechanism that is cognizant of network variability and performance. Network performance degrades as we move from HPC to cloud to edge, and may even fluctuate in shared clusters due to congestion, QoS (quality-of-service) priorities, contention, network scheduling, etc. In data-centers, network heterogeneity may even arise from device placement and

topology, link bandwidth, job placement, etc. The communication cost of lossy compression using Allgather to aggregation compressed gradient values and indices has varying latency and bandwidth cost compared to Allreduce used to aggregate the original, uncompressed gradients. [279] proposes heuristics based on latency-bandwidth cost model to choose the faster collective between Allgather and different flavors of Allreduce for a given network configuration. To leverage the latter collective, it implements an Allreduce-friendly compression technique called *AR-Topk*. For any CF, parallel and statistical efficiency in gradient compression are inversely related; a large CF improves parallel performance by reducing communication cost but may suffer poor statistical performance from trimming excessive gradient information. Similarly, a small CF would preserve statistical performance by retaining most of the tensors but have poor parallel efficiency due to high communication cost. To adaptively adjust the degree of compression under variable and unpredictable networks while accounting for parallel as well as statistical efficiency, [279] models compression as a multi-objective optimization problem. The objective of this optimization problem is to find the optimal CF $c_{optimal}$ that maximizes compression gain (extended from *GraVAC* [280]), and minimized compression and communication overhead: $c_{optimal} = \text{argmin}_c\{\mathcal{F}(\ t^{(c)}_{comp-decomp}, t^{(c)}_{sync}, gain^{-1(c)})\}$. In this evaluation, the authors show how *AR-Topk* dynamically adjusts its CF as well as communication collective as latency and bandwidth changes over the course of training.

5. Federated Learning

The challenges of distributed model training are further exacerbated in *Collaborative* or *Federated Learning (FL)* [155], [194], [295], [171] where numerous edge or mobile devices collect multi-modal data to be used for training. For e.g., edge devices could be traffic or car cameras, smart sensors in private networks, fog servers, etc. while mobile devices include smartphones or tablet cameras, sensors and smart keyboards. However, unlike typical distributed data-parallel training where the data is moved to a central location and partitioned in an IID manner, data recorded by personal/restricted devices poses a privacy or security risk. Distributed training on non-IID inherently suffers from slower and poor convergence over balanced partitioning among workers [325], [277]. Additionally, synchronous training mechanisms are applicable to cloud and data-centers with high-speed interconnects and bandwidth. Comparatively, edge devices have limited network bandwidth or high latency due to geographically distributed workers. Lastly, workers participating in federated learning may have heterogeneous computational capability (i.e., systems heterogeneity) and so, may face due straggler slowdown on account of slower devices in synchronous training. With the explosion in big data and IoT, the number of smart devices has grown exponentially. The FL space has thus observed rapid developments in recent years.

A. State-of-the-art FL Frameworks

A variety of benchmarks and frameworks have been developed in recent years for training a globally-shared model with federated clients in simulations as well as real-world deployments. A well-rounded framework should be scalable and able to train over a swarm of devices. It should also leverage different communication protocols like MPI, gRPC (remote procedure call) and message queuing [157] to enable both cross-silo and cross-device FL. In cross-silo, clients are fewer and have more computational resources available, such as nodes in shared clusters, cloud or HPC. In cross-device FL, clients are resource, power or network-constrained devices, such as those found across mobile, edge or IoT networks.

A thorough FL framework should also implement privacy-preserving techniques such as differential privacy [45], homomorphic encryption [214] and secure aggregation [71]. Differential privacy [45] adds fixed noise to model updates to prevent gradient inversion attacks and prevents associating a data sample with a particular client in the presence of bad actors. Homomorphic encryption (HE) [214] allows clients to send their updates in an encrypted format to the central server, which aggregates client updates without decrypting the data. Although this imposes additional computational overhead on the server, it ensures that client updates cannot be leaked in case of unreliable or attack-prone servers. Secure aggregation [71] combines updates from various clients such that the contributions of any given client remains private. This can be achieved either by homomorphic encryption, or via secret sharing where model updates are split into multiple shares, and distributed in smaller pieces.

An end-to-end FL framework should also leverage compression techniques, such as those described in §4-B.

Additionally, it must also cater to challenges like limited compute and data heterogeneity among clients, which is prevalent in edge and mobile devices. For privacy-preserving ML, it is critical to maintain secure access and robust identity and access management protocols (such as Globus Compute [78]). Lastly, it should also encompass state-of-the-art training algorithms, and provide good abstractions to implement novel ones. We describe some FL techniques in §5-B.

With the above features in mind, we list existing FL frameworks, and highlight their respective strengths and weaknesses (listed in the order of publication year):

- *PySyft* [236] is an open-source, privacy-preserving ML library that protects data by using techniques like homomorphic encryption and secure multi-party computation. It integrates PyTorch within its framework, provides single and multi-node execution, and creates virtual clients for simulation runs. For communication, PySyft uses the WebSocket protocol [27] to aggregate client and server updates.

- *LEAF* [76] provides diverse IID and non-IID datasets across a variety of ML tasks and includes various performance metrics to measure the convergence speedup and quality of FL algorithms. However, it is available for simulation-only and implements multiple clients with a simplistic FL training loop. It also does not implement any additional privacy preserving features or compression-based techniques.

- *FedML* [130] is an open-source, Python-based framework that can train models written either in PyTorch or TensorFlow. It supports data heterogeneity, privacy-preserving methods, authentication and access management and real-world deployment (in addition to simulations). FedML also offers a web-based job submission and performance monitoring/logging UI (user interface). However, FedML currently lacks compression capability and only implements synchronous FL algorithms (no support for asynchronous communication currently).

- *TensorFlow Federated (TFF)* [70] is an open-source framework for applying ML on decentralized data. With its two-level programming model, users can implement some modules in TensorFlow and some with TFF's API. TFF additionally allows for data heterogeneity and privacy-preserving techniques. However, TFF is limited to simulations only over multi-GPU systems and not geared towards actual deployment in the field. Compression is also missing from TFF at the time of writing this article.

- *FLOWER* [67] is developed for both simulations and real-world FL job deployments. Using gRPC for communication, FLOWER support multi-language support where clients and server can be implemented in different programming languages (assuming the underlying model being trained is the same). In principle, one could thus train over an Android client written in Java and another iOS client in Swift language, while the server is running Python code. However, this comes at the cost of higher communication overhead from serializing-deserialization (ser-de) the buffers between clients and servers. It also supports data heterogeneity over a plethora of datasets, and provides interfaces for private training and authentication mechanisms. However, it currently does not support asynchronous training algorithms or compression mechanisms. In its evaluation, FLOWER scaled to over 1 million devices with 1000 concurrently training clients.

- *FedScale* [161] supports both simulations and actual deployment across heterogeneous clients with different computational capabilities and unbalanced data distribution. With its modular architecture, FedScale provides support for various synchronous and asynchronous federated algorithms, and integrates well into existing frameworks like TensorFlow and PyTorch. It also provides compression-support, privacy-preserving ML and authentication mechanisms. However, for cross-silo FL in HPC, FedScale does not currently offer direct support for MPI integration which may enable faster training with better latency-bandwidth costs over gRPC with HTTP/2.

- *NVFLARE (NVIDIA Federated Learning Application Runtime Environment)* [233] offers synchronous FL over heterogeneous data with features like privacy, authentication techniques and real-world deployment capability. For communication, the framework uses gRPC with protocol buffers (ProtoBufs) for message ser-de. The provided abstractions in the Controller programming API like *FLContext* and *FLComponent* ease the development of new FL algorithms. However, it currently lacks abstractions to implement asynchronous

training algorithms with in-built compression to reduce data, model or gradient communication volume.

- *MONAI (Medical Open Network for AI)* [77] targets ML applications for medical data and built atop PyTorch and PyTorch lightning [19]. It provides a plethora of pre-trained models on medical images and provides implementations of state-of-the-art federated algorithms. Backed by NVIDIA, MONAI is well integrated with NVFLARE [233].

- *APPFL (Argonne Privacy-Preserving Federated Learning)* [237], [176] is a framework developed by the Argonne National Laboratory, USA, and enables FL on heterogeneous data with support for synchronous and asynchronous training algorithms across both simulations and real-world deployments. Additionally, it integrates compressors like SZ2 [183] and SZ3 [321], and provides privacy-preserving ML with homomorphic encryption and differential privacy. Using Globus compute [78], [7], APPFL offers additional authentication and identity/access management support, as well providing FL function-as-a-service (FaaS) [175]. APPFL uses MPI and gRPC as client-side communication protocol and Globus compute [7] respectively.

B. Training Algorithms in FL

Here, we describe some of the more popular and novel federated training algorithms and techniques proposed in recent years:

- *Federated Averaging* or *FedAvg* [194] is a federated learning technique that leaves the data locally on each device and learns a globally shared model by infrequently aggregating locally computed updates. Workers thus train independently on their local data, and occasionally send their updates to a central location like a parameter server. The synchronization may be performed at the end of each epoch, or after certain iterations as specified by the user. FedAvg is akin to local-SGD described previously, but applied to a plethora of mobile devices and non-I.I.D. data distributions. Another difference in FedAvg is that it performs *weighted aggregation* when different workers process variable number of data points in an iteration. The variation in data processing may stem from systems heterogeneity among workers or non-uniform data volume across workers. Thus, when worker i processes d_i samples in an iteration such that a total of N workers cumulatively process mini-batch d, the model aggregation step on the PS becomes $w_{t+1} \leftarrow \sum_{i=1}^{N} d_i \cdot w_{i,t+1}/d$. Compared to synchronous SGD, FedAvg reduces communication cost by 10-100$\times$ for various DNNs.

- *FedProx* [172] tackles both systems and statistical heterogeneity in federated networks. It is a generalization and re-parameterization of FedAvg, while providing more robust convergence guarantees. FedProx generalizes for FedAvg by allowing variable amount of work on each device based on its compute capability and aggregate these partial solutions instead of dropping the results from the stragglers. Interestingly, FedProx adds a proximal term akin to elastic-averaging SGD's penalty term (Equation (8a)) to the loss function. This term is added to limit the impact of variable local updates on the per-worker solvers by reducing the deviation between the local and global model state. The proximal term also allows variable amount of local work due to systems heterogeneity. Thus, FedProx optimizes to minimize the loss function updated with an additional proximal term and infrequently aggregates model state from all workers on a central server during communication period. The authors show that under the assumption of bounded variance and a large enough penalty/proximal term, FedProx achieves the complexity as SGD of $\mathcal{O}(N/\epsilon + 1/\epsilon^2)$ for an ϵ-approximate solution.

- [311] developed a non-parametric Bayesian inference approach for FL in non-IID, heterogeneous data settings. Models are trained locally on a device and those per-worker parameters are matched across data sources via a posterior of *Beta-Bernoulli process (BBP)* and *Indian Buffet Process (IBP)* [271]. A BBP is a Bayesian non-parametric model that matches local to global parameters or create new global parameters. An assignment cost matrix is formulated given a collection of workers with locally-trained parameters. Matching assignments are then computed using the Hungarian algorithm [159] to infer global parameters across all workers. The chosen global parameters from all workers are concatenated to form new global model parameters.

- [325] uses a decentralized approach to improve statistical performance of FL on unbalanced and skewed data.

They suggest that accuracy reduction in non-IID training can be explained by weight divergence, or *Earth mover's distance (EMD)* between distribution of classes on a worker and the overall population distribution. Weight divergence can be measured as: $||w_{FedAvg} - w_{SGD}||/||w_{SGD}||$. When the data is balanced, weight divergence is small; and larger for non-IID data. During the synchronization step, the global weights are computed using a weighted average on the parameters of the individual workers based on the number of training samples processed by each. To improve distribution and training performance, they create a small data subset that is globally shared between all workers. The authors showed an accuracy improvement of 30% on CIFAR-10 while sharing only 5% data globally.

- [155] reduces upstream communication in FL (i.e., from workers to a central server) via *structured* and *sketched* updates. With structured updates, workers learn from a restricted space with a small number of variables either via low-rank approximation or random masking. In sketched updates, all model parameters are pushed in compressed form using a mix of quantization, random rotations and subsampling. The authors validate their approach on convolutional and recurrent neural network by reducing communication cost by two orders of magnitude.

- *FetchSGD* [234] solves the challenge of expensive communication overheads and slow convergence due to sparse client participation at any iteration and non-IID data in FL. It reduces communication using count sketching for model compression. Different sketches across workers can be combined to aggregate model updates. Additionally, sketching is linear and supports both momentum and error accumulation.

$$\mathcal{S}(\widehat{g}_t) = \sum_{i=1}^{N} \mathcal{S}(g_{i,t}) \tag{14a}$$

$$\text{Top-}k(\mathcal{U}(\mathcal{S}(\widehat{g}))) \approx \text{Top-}k(\widehat{g}) \tag{14b}$$

$$\triangle_t = \text{Top-}k(\mathcal{U}(\eta\mathcal{S}_t + \mathcal{S}_t^{error})) \tag{14c}$$

$$\mathcal{S}_{t+1}^{error} = \eta\mathcal{S}_t + \mathcal{S}_t^{error} - \mathcal{S}(\triangle_t) \tag{14d}$$

$$w_{t+1} = w_t - \triangle_t \tag{14e}$$

In the synchronization step, the PS can compute the sketch (using compression operator $\mathcal{S}(\cdot)$) of the aggregated gradient by simply summing the individual gradients because of the linear property of sketching (Equation (14a)). The sketching operator has a corresponding decompression operator $\mathcal{U}(\cdot)$ associated with it that returns an unbiased estimate of the aggregated gradient. Thus, most significant parameters can be approximated using Top-k compression [263] as shown in Equation (14b). Gradients for the current iteration are computed using sketched updates and the accumulated error (or residual gradients) in Equation (14c). The residual gradients to be used in the next iteration are calculated by subtracting the sketched update from $\triangle_t$ (Equation (14d)), followed by applying the SGD update.

- *Federated Momentum* or *FedMom* [138] reassessed the model averaging step in previous algorithms like FedAvg and FedProx, and formulated it as a gradient-based method with biased gradients. Instead of simple model averaging, FedMom updates model parameters as: $w_{t+1} = w_t - \sum_{i=1}^{N} d_k(w_t - w_{t+1}^{(i)})/d$. With this approach, FedMom updates its momentum vector in order to minimize client drift (i.e., divergence between local client models and aggregated server model). The momentum coefficient is a tunable hyperparameter that limits the divergence between clients.

- *SCAFFOLD* [150] leverage variance reduction in order to minimize the client drift in local updates. Stochastic controlled averaging helps DNNs converge in fewer communication rounds, even with heterogeneous data distribution. With control variates on a per-client basis, SCAFFOLD accounts for local model divergence as well as differences in clients' data distribution. For client i with local model $w_t^{(i)}$ and gradient update $g_t^{(i)}$, parameters are updated as shown in Equation (15a). The control variate is updated by the difference between

the global and local model state (Equation (15b)). After computing the local client updates, the global model is updated as a gradient update by calculating the average difference between clients' local state between the current and previous iteration t, shown in Equation (15c).

$$w_{t+1}^{(i)} = w_t^{(i)} - \eta \odot (g_t^{(i)} + c^{(i)}) \tag{15a}$$

$$c_{t+1}^{(i)} = c_t^{(i)} + (w_t^{(i)} - w_t) \tag{15b}$$

$$w_{t+1} = w_t - \frac{\eta}{N} \odot \sum_{i=1}^{N} (w_{t+1}^{(i)} - w_t^{(i)}) \tag{15c}$$

- *Federated Normalized Averaging* or *FedNova* [293] studies the objective inconsistency arising from heterogeneity in local data and computational speed across workers. In FedNova, the authors make the following assumptions: each local objective function is L-smooth, local gradient on each worker is unbiased and has a bounded variance, and the dissimilarity between the weights of any of the participating workers is bounded as well. Under these assumptions, the non-vanishing error term due to objective inconsistency $\mathcal{X}^2_{d||w} = \sum_{i=1}^{N}(d_i - w_i)^2/w_i$, where d_i is the fraction of data points processed in a given iteration on worker i relative to the global mini-batch size. This term measures the chi-square divergence between the model parameters computed on a worker based on the quantity of training data processed by it. FedAvg eliminates non-vanishing error resulting from objective inconsistency by setting $w_i = d_i$. FedNova thus aggregates the normalized stochastic gradients instead of the local gradient changes. The authors show how FedAvg and FedProx can be presented as special cases of the analysis framework included with FedNova.

- *ScaDLES* [277] enables federated learning over streaming data. It proposes weighted aggregation of updates based on a worker's respective streaming rate in order to mitigate compute and streaming heterogeneity. To mitigate statistical heterogeneity arising from non-IID data streaming locally into devices, the authors propose randomized *data-injection* where only a device shares partial training samples with a subset of devices chosen randomly. On each iteration, a device i chooses a fraction α out of N devices to share fraction β of its streaming data, βS_i. Together, (α, β) decide what fraction of workers share how much of their data with other devices. ScaDLES minimizes privacy violation (due to data sharing) by randomly choosing devices and broadcasting only a fraction of the data.

- *FedBN* [173] deals with the challenge of different workers storing data with different distributions locally, or *feature-shift non-IID*. This approach uses local batch normalization [139] to mitigate feature shift before the synchronization period. FedBN outperforms both FedAvg and FedProx by achieving 6-10% more accuracy on various non-IID datasets like Office-Caltech-10 [15] and DomainNet [212].

- *SelSync* [278] is a low-frequency, high-volume communication-based federated learning technique that adaptively switches between local and synchronous updates based on the significance of its gradient updates. The significance of model updates is measured via the gradient change metric given as: $\left|\frac{\mathbb{E}[||\nabla\mathcal{L}_{(t)}^{(c)}||^2] - \mathbb{E}[||\nabla\mathcal{L}_{(t-1)}^{(c)}||^2]}{\mathbb{E}[||\nabla\mathcal{L}_{(t-1)}^{(c)}||^2]}\right|$ for gradients $\nabla\mathcal{L}$ computed over consecutive iterations t and $(t-1)$. Using the above metric, updates are applied locally if its value is below the user-specified threshold, and synchronous updates otherwise. Additionally, SelSync employs parameter aggregation over gradient averaging and proposes a custom data-partitioning scheme ideal for semi-synchronous training. It also extends stochastic data-injection from ScaDLES [277] to improve convergence quality in skewed and non-IID data settings.

- *AdaFed* [115] takes an adaptive approach to federated learning by dynamically weighting local worker replicas in the parameter aggregation step, as well as adapting the loss function of local solvers in the communication period. Due to its adaptive behavior, AdaFed is robust to unbalanced and non-IID data. The weights of each worker's replica is dynamically assigned on the basis of the local model's performance on a representative

test set, where more weights are assigned to workers with better performance on the test set. Additionally, each worker's loss function is updated as well based on the central server's performance on the test set. The updated model parameters and loss function is then propagated back to the workers from the PS.

- *Distributed Low-Communication* or *DiLoco* [100], [140] trains LLMs in federated settings, and implements a variation of federated averaging [194] with only a few communication rounds to reach convergence. For LLM training, DiLoCo trains local clients over Adam [153] with weight decay optimization [192], while client updates over the central server are aggregated with a momentum-based optimizer. For fixed cumulative iterations, DiLoCo convergence to acceptable loss whether training from scratch or pre-training for certain iteration beforehand. The open-source version of DiLoCo is available at [140].

- *FedOpt* [226] proposes federated versions of adaptive optimizations like Adagrad [103], Adam [153] and Yogi [314]. In general, FedOpt computes the local parameters on a worker with its private data and evaluates the parameter changes between the local and global state in the communication step, i.e., $\triangle w_{i,t} = w_{i,t} - w_{*,t}$ for worker i at iteration t. These parameter changes are sent to the central server to be aggregated, which then re-distributes them back to the workers to be used in the next round of iterations: $\triangle w_t = \sum_{i=1}^{N} \triangle w_{i,t}/N$. Conventional adaptive gradient optimizers are extended to FedOpt to compute the momentum using the change in model parameters (i.e., $\triangle w_t$): $m_t = \beta_1 m_{t-1} + (1-\beta_1)\triangle w_t$. For *FedAgagrad*, velocity is computed as $v_t = v_{t-1} + \triangle w_t^2$; for *FedAdam* $v_t = \beta_2 v_{t-1} - (1-\beta_2)\triangle w_t^2$, and for *FedYogi* $v_t = v_{t-1} - (1-\beta_2)\triangle w_t^2 \cdot \text{sign}(v_{t-1} - \triangle w_t^2)$. Using the momentum and velocity terms m_t and v_t respectively, the parameter update is given as $w_{*,t+1} = w_{*,t} + \eta \frac{m_t}{\sqrt{v_t}+\tau}$.

6. Discussion

The first quarter of the 21st century has witnessed breakthroughs in the field of DL and AI (artificial intelligence) with various algorithmic innovations. As-per the scaling hypothesis [9], state-of-the-art models are built on a scalable architecture by stacking multiple units together, such as fully connected layers in multi-layer perceptron [232], convolutional layers in deep convolutional networks [163], [158], [254], skip-connection based residual networks [131] and large-language models built from attention-based transformers [283], [149], [95], [89], [39], [34], [35]. Although scaling up these architectures gives highly accurate and generalized models, this comes at the cost of high computational cost in training/inference. From Sutton's bitter lesson [20], greater AI systems are built by scaling computation and learning model parameters iteratively and repeatedly, rather than solely developing new ways to embed human knowledge into new compute agents.

The processing speed in von Neumann architecture [59] is governed by how fast and efficiently instructions are executed, along with data-transfer speeds from memory store to compute agent over the bus. Thus, each of the three components: compute, memory and interconnect are critical for optimal execution. In the context of DL and AI, output is some input-dependent prediction, and the learning task is to obtain the latent features or model parameters. *Compute efficiency* can be improved by executing more operations per-unit of time, which may correspond to floating-point operations per second (FLOPS) or MAC operations. As training is iterative and composed of many repetitive iterations, compute in the context of DL may also imply the total computation cost of training a model to convergence. DNN training also needs considerable memory to accommodate batches of training data, model parameters, gradients, activation maps and optimization-specific information. A slow interconnect may pose transmission delays between memory and compute that could lead to application stalling. The 'compute' component such as the CPU, or the memory at different hierarchy levels or interconnects like buses and network interfaces represent different application bottlenecks. For DL tasks, the development of specialized accelerators like GPUs, TPUs and FPGAs (field programmable gate arrays) have significantly accelerated compute and rendered data-movement (such as across memory and interconnects) as the primary performance bottleneck instead [98]. Thus, along with algorithmic innovations, significant compute gains in commodity hardware [199], wide availability of low-cost resources and HPC have aided the development and training of state-of-the-art DNNs.

A. Parallel Efficiency in Deep Learning

Along with the growing size and complexity of DL models, big data and IoT have resulted in exponential growth of multimodal data, thus making it crucial to integrate DL with distributed computing. This involves running calculations across a cluster of machines to achieve a common objective [73]. Instead of a single compute unit, a network of interconnected devices collectively solve large and complex tasks while providing scalability, consistency, transparency, availability and efficiency. Today, frameworks like Hadoop MapReduce [90], Apache Spark [313] and Apache Storm [272], and parallelization libraries like MPI [109], [17], OpenMP [88] and OpenSHMEM [18] provide easy abstractions for running large-scale computational problems. Running DL models on distributed systems presents similar challenges to those observed in a typical von Neumann architecture; distributed deep learning incurs considerable cost due to data-movement (from inter/intra-node communication and I/O overheads). It is crucial to optimize these components so as to improve the parallel efficiency of distributed DL. 'Compute' is improved by hardware-software co-design (described in §2-A), different parallelization strategies such as data, model, pipeline or tensor-parallelism (§2-B) 'Communication' is improved at both intra and inter-node levels; the former by using caching, optimized memory access patterns and tiling [8], using lower precision formats [196], [224], pipelining, asynchronous data movement [2], large-batch training [193], NUMA (non uniform memory access) aware task placement [174], and and optimized libraries for intra-node communication [88], [18], [31]. Inter-node communication is improved by using higher-bandwidth interconnects and fast communication libraries like RDMA and RDMA over converged ethernet (RoCE) [122], [113], topology-aware communication [56] and optimized aggregation like hierarchical [281] or ring-allreduce [242], overlapping computation with communication [225] and various compression techniques described in §4. Furthermore, as Moore's law [199] has trickled down towards edge computing, it has facilitated collaborative or federated learning to train DNNs while conscient of network bandwidth, data privacy and locality.

B. Statistical Efficiency in Deep Learning

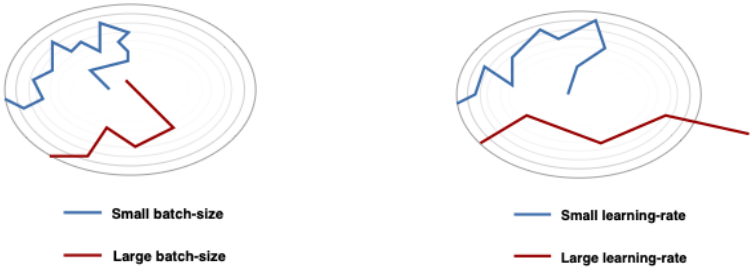
DNN training primarily entails learning a specific optimization landscape for a given loss function and dataset. Model hyperparameters set prior or tuned during training tend to affect its statistical efficiency, which in turn affects the overall convergence quality (like test/validation loss, accuracy, perplexity, bilingual evaluation understudy or BLEU score, etc.). Parallelizing DNNs across multiple nodes imposes both parallel and statistical efficiency considerations and must be studied to attain fast speedups and good generalization.

Choosing the optimal parameters is crucial for convergence, whether for highest quality model, or least training time. The statistical performance of a DNN is determined by the duration of training (total iterations or epochs), learning-rate schedule, weight decay, dropout and regularization, choice of activation function for non-linearity (e.g., sigmoid, ReLU, Tanh, etc.) [160], training batch-size, loss function used (mean squared error, cosine similarity, cross-entropy, KL divergence, etc.) [294], optimization type (e.g., Nesterov's momentum, Adam, Adagrad, RMS-prop, AdamW, etc.) [84] or weight initialization (such as Xavier or He). We explain some of them here:

- *Length of training:* In well conditioned optimization problems, learnable parameters follow the direction of low curvature where loss decreases in a predictable way. Repeating this iteratively over the iterations improves a DNNs statistical performance. However, in poor and sub-optimal conditions, models may not necessarily converge in an expected manner as it may saturate to a saddle point. Thus, training for longer duration may prove ineffective in such scenarios.

- *Type of model optimization:* Apart from vanilla SGD update, other optimization techniques can learn the model faster or towards a flatter minima over the non-convex loss landscape. For e.g., *Nesterov* [204] adds a momentum-term to the gradients, essentially applying smoothing and thereby reducing noise. *Adagrad* or adaptive gradient [103] tunes the learning-rate for every parameter by considering the summed outer product of gradients from previous iterations. *RMS-Prop* [84] extends *Adagrad* by adjusting per-parameter learning-rate using exponential smoothing rather than cumulative sum. *Adam* or adaptive moment [153], and *Adamax* combine multiple optimization schemes by smoothing out the gradients' from previous iterations and decaying average of gradient variance to account for bias-correction from diminishing or sparse gradients.

- *Training batch-size:* Although training is faster with mini-batch GD compared to full-batch GD, the quality



(a) Batch-size (b) Learning-rate

Fig. 2: Optimization trajectory with (a) Small and large batch-sizes over the loss landscape. Large batches tend to have less noise and attain optimal training loss in bigger steps and fewer iterations. (b) Low and high learning-rates. A large step-size may overshoot and diverge from the minima.

of estimated gradients varies depending on the mini-batch size. Gradients are approximated to the true loss landscape depending on the chosen batch-size, and are inversely proportional to gradient noise (i.e., smaller batches have more noisy gradients) [193], [275]. On the other hand, gradients calculated from larger-batches are representative of the true gradients from full-batch GD. This is the one of the main benefits of weak-scaling in synchronous, data-parallel training where computations are parallelized with different batches over multiple devices to effectively produce large batches for GD update. From Figure (2a), large batches can achieve low training loss by taking bigger (and fewer) steps over the optimization landscape [307], [185]. Comparatively, small-batches are more noisy and thus, take many more steps to arrive at the minima. Despite its parallel efficiency and good training performance, large-batches suffer from poor generalization [152], [308] because they either overfit or converge to sharper minima. Prior works have thus explored studying gradient information to find the optimal batch-size that balances both parallel and statistical efficiency in DL [193], [275].

- *Learning-rate schedule:* In vanilla GD, gradients are scaled by the learning-rate, a crucial hyperparameter that may determine model convergence or divergence. From Figure (2b), a low step-size takes smaller steps and gradually moves in the direction of minima [193]. A higher learning-rate has a tendency to overshoot while taking larger steps and may in turn, diverge the model. Dynamic or adaptive learning-rate routines [315] that periodically decay the step-size by some factor aim to mitigate such issues [245]. Such adaptive schemes can tune learning-rates at different granularities, such as across entire model, layer-wise or parameter-wise [315], [310]. The intuition behind these techniques is that as a DNN gradually converges and fine-tunes itself, its gradients saturate and so the factor by which SGD update is applied needs to be adjusted as well. Training techniques like learning-rate warmup [119] have also been mapped to gradient compression, where a small CF is used in the beginning and increased uniformly through the course of training [187], [50]. In the learning-rate warmup phase, step-size is scaled as η/I for first I iterations, and set to η afterwards. The initial, small values (either in learning-rates or CF in compression) account for the early sensitive training phase [111], [47].

Distributing a DL job over multiple nodes or devices offers parallel gain, or statistical gain, or both. This comes from either training large models (like LLMs) that are too big to fit or run on a single machine, or training DNNs over massive datasets that must be executed in a distributed manner in order to converge within reasonable times. In the former, parallelization strategies like model, pipeline and tensor parallelism enable training of massive DNNs divided across multiple machines (with high resource utilization). For the latter, synchronous data-parallel training applies weak-scaling, i.e., increases the global batch-size proportionally to the number of nodes. Thus, if a single node processes b samples at each iteration, synchronous data-parallelism processes a global batch-size of Nb across N nodes. From a statistical standpoint, DL is affected by factors like gradient noise, learning-rate routines, model initialization and poor conditioning. On top of these challenges, distributed DL is also affected by heterogeneous hardware and architectures, heterogeneous networks, skewed and unbalanced distribution of training data and generalization degradation with large-batch training. It is thus crucial to consider all these factors for scalable DL systems that are time-and-compute optimal, and produce highly generalizable and effective models.

7. Current and Future Directions

In the last quarter century, we have moved through the digital, information and data age, and now into the era where compute is king [134], given the amount of training and inference compute needed to develop or serve state-of-the-art models. Computation is the new oil today, which is why countries are imposing sanctions on export of chips [23], and why many organizations rush to build in-house AI supercomputing clusters [28], [11], [29]. LLMs today are driving us towards much higher computational demands of Artificial General Intelligence (AGI); reasoning models today that are served to the masses are spending as much compute on inference as training, if not more. Compute today follows Jevons' paradox [53]; even though compute has become cheaper and accessible, the efficacy of DNNs today has increased usage and demand, driving up the overall computational demands.

From multi-layer perceptrons (MLP) to LLMs, DNNs are compute-intensive structures. The attention block of a basic transformer [283] accounts for 95% of its operations in matrix multiplication. From there, size and complexity of DNNs grew further with models like BERT [95], GPT-2 [218], GPT-3 [149], [74], ChatGPT with Reinforcement Learning with Human Feedback (RLHF) and Direct Preference Optimization (DPO) [219], GPT-3.5 [38], LLaMA [35], and many more. As we pace towards trillion parameter models, compute cost will rise exponentially. Optimizing the compute cost, quantizing these models, minimizing data movement, developing faster and cheaper communication infrastructure, and building efficient algorithms will thus determine the future of AI.

In addition to compute scaling, another observed trend is *data scaling* [134] Current state-of-the-art models have already been trained over internet-scale knowledge, and it is estimated that we will eventually run out of data to train on [148]. A potential solution to this approach is to use synthetic data generated by LLMs to train new AI systems [41], [191]. With data scaling, we want to train over better data and build more robust and effective models. Akin to LLMs, it is equally important to train smaller models with more data (i.e., invest in pre-training compute) in order to have small, generalizable models with low inference costs, or develop context-aware models that run locally on mobile and edge devices. In FL context, there are avenues for further developments as most of the data is secure and private, i.e., unseen by a pre-trained model. Training DNNs in a privacy-preserving manner over distributed clients with potential computational heterogeneity warrants development of novel algorithms and techniques to address both parallel and statistical challenges.

It is also predicted that pre-training as we know it will end for generic AI applications [40], as the fossil fuel for AI, *open* training datasets are limited to current state of the internet. For specific use cases, fine-tuning and post-training will play an important role to enhance models for specific tasks. Techniques like Low-Rank Adaptation (LoRA) [136] enable parameter-efficient post-training and reduce the inference cost for task-specific learning. Further advancing such methods would help decompose even larger models for edge or IoT deployment.

The efficacy of reinforcement learning in the context of DNNs was firmly established almost a decade ago with AlphaGo [252] and AlphaZero [253]. Later, reinforcement techniques like RLHF and DPO [219] further enhanced the capability and generalizability of LLMs. *Language reasoning models (LRMs)* [66] can already perform better than majority of the human programmers [128] and *STEM* (science, technology, engineering, mathematics) researchers in some cases [42]. To proceed into the age of reasoning with LRMs, it is imperative to combine and optimize Reinforcement Learning (RL) strategies and currently existing, internet-scale knowledge models with HPC resources [66]. Here, LRMs are trained with RL where the policy and value functions will be the current state-of-the-art LLMs [66].

DNNs training in either centralized or decentralized settings incurs significant compute cost, both for training and inference serving. Companies like NVIDIA and AMD are aggressively competing in the chip manufacturing space to build state-of-the-art accelerators, while companies like Meta, Google, Tesla and Microsoft are scaling to build the next generation of supercomputers in the form of AI clusters. Despite the stock market crash in January 2025 [3] from DeepSeek's release of a cohort of open-source LLMs and LRMs with significantly lower training cost in terms of the $ amount, it is expected that cost of AI training and inference will rise significantly in coming years. Ignoring these short-term fluctuations, the size, computational requirements, training cost (in terms of time, money and energy) will rise in the long-term. Building efficient computational and communication strategies that accelerate training and inference will be critical to develop scalable AI systems, and crucial for the future of AI in general.

About the author

Sahil Tyagi is a researcher broadly working at the intersection of AI and computer systems. He obtained a Ph.D. in Intelligent Systems Engineering (ISE) at the Luddy School of Informatics, Computing and Engineering from Indiana University Bloomington. There, he worked at the intersection of deep learning, distributed and ML systems, and his thesis focused on developing efficient computational and communication models to scale DNNs across edge, cloud and HPC. Currently, he is a postdoctoral research associate at Oak Ridge National Laboratory, USA, where he is working on topics like distributed training, federated learning, compression, communication optimization and privacy-preserving ML. His recent works can be found *here*. He can be contacted at tyagis@ornl.gov.

References

[1] Anthropic: Claude Sonnet 3.5.
[2] CUDA Programming Guide https://docs.nvidia.com/cuda/cuda-c-programming-guide/.
[3] DeepSeek sparks AI stock selloff; NVIDIA posts record market-cap loss. link: https://www.reuters.com/technology/chinas-deepseek-sets-off-ai-market-rout-2025-01-27/.
[4] DeepSpeed GitHub.
[5] Facebook Caffe2 blog: https://research.facebook.com/downloads/caffe2/.
[6] Facebook Gloo https://github.com/facebookincubator/gloo.
[7] Globus Compute.
[8] GPU Memory levels https://cvw.cac.cornell.edu/gpu-architecture/gpu-memory/memory_levels.
[9] Gwern: The Scaling Hypothesis https://gwern.net/scaling-hypothesis.
[10] Microsoft DMTK documentation: https://github.com/microsoft/dmtk.
[11] Microsoft has purchased half a million advanced NVIDIA chips as part of AI investment: https://www.grip.globalrelay.com/microsoft-has-purchased-half-a-million-advanced-nvidia-chips-as-part-of-ai-investment/.
[12] NVIDIA cuBLAS https://developer.nvidia.com/cublas.
[13] NVIDIA GPU Memory Hierarchy and DL Performance.
[14] NVIDIA Tesla K80 GPU specsheet https://www.nvidia.com/content/dam/en-zz/solutions/data-center/tesla-product-literature/tesla-k80-boardspec-07317-001-v05.pdf.
[15] Office-Caltech10 dataset.
[16] OpenAI o1 model.
[17] OpenMPI https://www.open-mpi.org.
[18] OpenSHMEM: http://openshmem.org/site/.
[19] PyTorch Lightning.
[20] Rich Sutton: The Bitter Lesson http://www.incompleteideas.net/incideas/bitterlesson.html.
[21] ROCm HiP.
[22] The Claude 3 Model Family: Opus, Sonnet, Haiku.
[23] This is how China skirts U.S. chip bans.
[24] Torch: A Scientific Computing Framework for Machine Learning documentation http://torch.ch/.
[25] TorchMobile for Android and iOS documentation https://pytorch.org/mobile/home/.
[26] TorchServe documentation https://pytorch.org/serve/.
[27] WebSocket Protocol.
[28] xAI Colossus: https://www.supermicro.com/en/featured/xai-colossus.
[29] Zuckerberg's Meta is Spending Billions to Buy 350,000 NVIDIA H100 GPUs: https://www.pcmag.com/news/zuckerbergs-meta-is-spending-billions-to-buy-350000-nvidia-h100-gpus.
[30] *Reevaluating Amdahl's Law*. IEEE Computer Society Press, 1994.
[31] NCCL: NVIDIA Collective Communication Library, 2015.
[32] AI and Compute, https://openai.com/blog/ai-and-compute, 2018.
[33] Microsoft blog. Turing-NLG: A 17B parameter language model by Microsoft, 2020.
[34] Google BARD https://blog.google/technology/ai/bard-google-ai-search-updates/, 2023.
[35] Meta's LLaMA https://ai.facebook.com/blog/large-language-model-llama-meta-ai/, 2023.
[36] OpenAI ChatGPT Blog: https://openai.com/blog/chatgpt, 2023.
[37] OpenAI ChatGPT Statistics https://lambdalabs.com/blog/demystifying-gpt-3, 2023.
[38] OpenAI GPT-3.5 https://platform.openai.com/docs/models/gpt-3.5-turbo, 2023.
[39] OpenAI GPT-4 https://openai.com/research/gpt-4, 2023.
[40] Ilya Sutskever: "Sequence to sequence learning with neural networks: what a decade". link: https://www.youtube.com/watch?v=1yvbqashlzs, 2024.
[41] LLM synthetic data: pros, cons, and how to merge it with human data. Link: https://www.superannotate.com/blog/llm-synthetic-data, 2024.
[42] How to Build an AI Agent for Research. 2025.
[43] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, and A. H. et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, 2016.

[44] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Z. et al. TensorFlow: A System for Large-Scale Machine Learning, 2016.

[45] M. Abadi, A. Chu, I. J. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep Learning with Differential Privacy. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[46] A. M. Abdelmoniem, A. M. Elzanaty, M.-S. Alouini, and M. Canini. An Efficient Statistical-based Gradient Compression Technique for Distributed Training Systems. *ArXiv*, abs/2101.10761, 2021.

[47] A. Achille, M. Rovere, and S. Soatto. Critical Learning Periods in Deep Neural Networks, 2019.

[48] G. Adam, N. Chris, P. Josh, W. Melanie, D. BlackAlex, K. Vyacheslav, A. Samuel, and E. E. Susan. Deeplearning4J: Distributed, open-source deep learning for Java and Scala on Hadoop and Spark. 2016.

[49] A. Agarwal and J. C. Duchi. Distributed Delayed Stochastic Optimization, 2011.

[50] S. Agarwal, H. Wang, K. Lee, S. Venkataraman, and D. Papailiopoulos. Accordion: Adaptive Gradient Communication via Critical Learning Regime Identification, 2020.

[51] S. Agarwal, H. Wang, S. Venkataraman, and D. Papailiopoulos. On the Utility of Gradient Compression in Distributed Training Systems, 2021.

[52] A. F. Aji and K. Heafield. Sparse Communication for Distributed Gradient Descent. In *Proceedings of the 2017 Conference on Empirical Methods in Natura Language Processing*. Association for Computational Linguistics, 2017.

[53] B. Alcott. Jevons ' paradox. 2005.

[54] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding, 2017.

[55] D. Alistarh, T. Hoefler, M. Johansson, S. Khirirat, N. Konstantinov, and C. Renggli. The Convergence of Sparsified Gradient Methods, 2018.

[56] M. Amaral, J. Polo, D. Carrera, S. R. Seelam, and M. Steinder. Topology-Aware GPU Scheduling for Learning Workloads in Cloud Environments. *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[57] K. H. Ang, G. Chong, and Y. Li. PID control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, 13(4):559–576, 2005.

[58] F. Bach and E. Moulines. Non-Asymptotic Analysis of Stochastic Approximation Algorithms for Machine Learning. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, page 451–459, Red Hook, NY, USA, 2011. Curran Associates Inc.

[59] J. W. Backus. Can programming be liberated from the von Neumann style?: A Functional style and its Algebra of programs. *Commun. ACM*, 21:613–641, 1978.

[60] Y. Bao, Y. Peng, Y. Chen, and C. Wu. Preemptive All-reduce Scheduling for Expediting Distributed DNN Training. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 626–635, 2020.

[61] P. Baran. On Distributed Communications: Introduction to Distributed Communications Networks. 1964.

[62] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum Learning. In *International Conference on Machine Learning*, 2009.

[63] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[64] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar. signSGD: Compressed Optimisation for Non-Convex Problems, 2018.

[65] J. Bernstein, J. Zhao, K. Azizzadenesheli, and A. Anandkumar. signSGD with Majority Vote is Communication Efficient And Fault Tolerant, 2019.

[66] M. Besta, J. Barth, E. Schreiber, A. Kubicek, A. C. Catarino, R. Gerstenberger, P. Nyczyk, P. Iff, Y. Li, S. Houliston, T. Sternal, M. Copik, G. Kwa'sniewski, J. Muller, L. Flis, H. Eberhard, H. Niewiadomski, and T. Hoefler. Reasoning Language Models: A Blueprint. *ArXiv*, abs/2501.11223, 2025.

[67] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, T. Parcollet, and N. D. Lane. Flower: A Friendly Federated Learning Research Framework. 2020.

[68] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang, M. Pieler, U. S. Prashanth, S. Purohit, L. Reynolds, J. Tow, B. Wang, and S. Weinbach. GPT-NeoX-20B: An Open-Source Autoregressive Language Model, 2022.

[69] M. Blot, D. Picard, and M. Cord. GoSGD: Distributed Optimization for Deep Learning with Gossip Exchange, 2018.

[70] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konecný, S. Mazzocchi, H. B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander. Towards Federated Learning at Scale: System Design. *ArXiv*, abs/1902.01046, 2019.

[71] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical Secure Aggregation for Privacy-Preserving Machine Learning. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[72] F. Brakel, U. Odyurt, and A.-L. Varbanescu. Model Parallelism on Distributed Infrastructure: A Literature Review from Theory to LLM Case-Studies, 2024.

[73] T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. A. Hensgen, and R. F. Freund. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *J. Parallel Distributed Comput.*, 61:810–837, 2001.

[74] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. teusz Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language Models are Few-Shot Learners. *ArXiv*, abs/2005.14165, 2020.

[75] E. Buber and B. DIRI. Performance Analysis and CPU vs GPU Comparison for Deep Learning. In *2018 6th International Conference on Control Engineering and Information Technology (CEIT)*, pages 1–6, 2018.

[76] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečný, H. B. McMahan, V. Smith, and A. Talwalkar. LEAF: A Benchmark for Federated Settings, 2019.

[77] M. J. Cardoso, W. Li, R. Brown, and N. M. et al. MONAI: An open-source framework for deep learning in healthcare. *ArXiv*, abs/2211.02701, 2022.

[78] R. Chard, Y. Babuji, Z. Li, T. J. Skluzacek, A. Woodard, B. Blaiszik, I. T. Foster, and K. Chard. funcX: A Federated Function Serving Fabric for Science. *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, 2020.

[79] C.-C. Chen, C.-L. Yang, and H.-Y. Cheng. Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-GPU Platform, 2019.

[80] C.-Y. Chen, J. Choi, D. Brand, A. Agrawal, W. Zhang, and K. Gopalakrishnan. AdaComp : Adaptive Residual Gradient Compression for Data-Parallel Distributed Training, 2017.

[81] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *ArXiv*, abs/1512.01274, 2015.

[82] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing Neural Networks with the Hashing Trick, 2015.

[83] Y. Chen, K. Yuan, Y. Zhang, P. Pan, Y. Xu, and W. Yin. Accelerating Gossip SGD with Periodic Global Averaging. In *International Conference on Machine Learning*, 2021.

[84] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl. On Empirical Comparisons of Optimizers for Deep Learning. *ArXiv*, abs/1910.05446, 2019.

[85] F. Chollet. Keras for quick ML prototyping: https://github.com/fchollet/keras. https://github.com/fchollet/keras, 2015.

[86] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. Solving the Straggler Problem with Bounded Staleness. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, Santa Ana Pueblo, NM, May 2013. USENIX Association.

[87] C. A. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. D. Bailis, K. Olukotun, C. Ré, and M. A. Zaharia. DAWNBench : An End-to-End Deep Learning Benchmark and Competition. 2017.

[88] L. Dagum and R. Menon. OpenMP: An industry standard API for Shared-Memory Programming. 1998.

[89] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov. Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context, 2019.

[90] M. Dayalan. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51:107–113, 2008.

[91] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large Scale Distributed Deep Networks. In *NIPS*, 2012.

[92] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao. Optimal Distributed Online Prediction Using Mini-Batches. *J. Mach. Learn. Res.*, 13(null):165–202, jan 2012.

[93] P. Delestrac, J. Miquel, D. Bhattacharjee, D. Moolchandani, F. Catthoor, L. Torres, and D. Novo. Analyzing GPU Energy Consumption in Data Movement and Storage. *2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 143–151, 2024.

[94] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[95] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *ArXiv*, abs/1810.04805, 2019.

[96] S. Di and F. Cappello. Fast Error-Bounded Lossy HPC Data Compression with SZ. *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 730–739, 2016.

[97] E. Diao, G. Wang, J. Zhan, Y. Yang, J. Ding, and V. Tarokh. Pruning Deep Neural Networks from a Sparsity Perspective, 2023.

[98] J. Dongarra. ACM A. M. Turing Award Lecture SC22. In *Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis (SC)*, 2022.

[99] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, 2021.

[100] A. Douillard, Q. Feng, A. A. Rusu, R. Chhaparia, Y. Donchev, A. Kuncoro, M. Ranzato, A. Szlam, and J. Shen. DiLoCo: Distributed Low-Communication Training of Language Models. *ArXiv*, abs/2311.08105, 2023.

[101] N. Dryden, S. A. Jacobs, T. Moon, and B. Van Essen. Communication Quantization for Data-Parallel Training of Deep Neural Networks. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*, MLHPC '16, page 1–8. IEEE Press, 2016.

[102] N. Du, Y. Huang, A. M. Dai, S. Tong, D. Lepikhin, Y. Xu, M. Krikun, Y. Zhou, A. W. Yu, O. Firat, B. Zoph, L. Fedus, M. Bosma, Z. Zhou, T. Wang, Y. E. Wang, K. Webster, M. Pellat, K. Robinson, K. Meier-Hellstern, T. Duke, L. Dixon, K. Zhang, Q. V. Le, Y. Wu, Z. Chen, and C. Cui. GLaM: Efficient Scaling of Language Models with Mixture-of-Experts, 2022.

[103] J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.*, 12(null):2121–2159, jul 2011.

[104] D.-A. et al. DeepSeek-V3 Technical Report. *ArXiv*, abs/2412.19437, 2024.

[105] D.-A. et al. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *ArXiv*, abs/2501.12948, 2025.

[106] Ö. Eğecioğlu, Çetin Kaya Koç, and A. J. Laub. A recursive doubling algorithm for solution of tridiagonal systems on hypercube multiprocessors. 1989.

[107] J. Fang, H. Fu, G. Yang, and C.-J. Hsieh. RedSync : Reducing Synchronization Traffic for Distributed Deep Learning. *J. Parallel Distributed Comput.*, 133:30–39, 2018.

[108] N. S. Ferdinand, H. Al-Lawati, S. C. Draper, and M. S. Nokleby. Anytime Minibatch: Exploiting Stragglers in Online Distributed Optimization. *ArXiv*, abs/2006.05752, 2020.

[109] M. P. I. Forum. MPI: A Message-Passing Interface standard. 1994.

[110] M. Franceschelli, A. Giua, and C. Seatzu. Load balancing over heterogeneous networks with gossip-based algorithms. In *2009 American Control Conference*, pages 1987–1993, 2009.

[111] J. Frankle, D. J. Schwab, and A. S. Morcos. The Early Phase of Neural Network Training, 2020.

[112] M. Ganaie, M. Hu, A. Malik, M. Tanveer, and P. Suganthan. Ensemble Deep Learning: A Review. *Engineering Applications of Artificial Intelligence*, 115:105151, oct 2022.

[113] A. Gangidi, R. Miao, S. Zheng, S. J. Bondu, G. Goes, H. Morsy, R. Puri, M. Riftadi, A. J. Shetty, J. Yang, S. Zhang, M. J. Fernandez, S. Gandham, and H. Zeng. RDMA over Ethernet for Distributed Training at Meta Scale. ACM SIGCOMM '24, page 57–70, New York, NY, USA, 2024. Association for Computing Machinery.

[114] A. L. Gaunt, M. A. Johnson, M. Riechert, D. Tarlow, R. Tomioka, D. Vytiniotis, and S. Webster. AMPNet: Asynchronous Model-Parallel Training for Dynamic Neural Networks, 2017.

[115] A. Giuseppi, L. Della Torre, D. Menegatti, and A. Pietrabissa. AdaFed: Performance-Based Adaptive Federated Learning. In *Proceedings of the 5th International Conference on Advances in Artificial Intelligence*, ICAAI '21, page 38–43, New York, NY, USA, 2022. Association for Computing Machinery.

[116] A. Glaese, N. McAleese, M. Trebacz, J. Aslanides, V. Firoiu, T. Ewalds, M. Rauh, L. Weidinger, M. Chadwick, P. Thacker, L. Campbell-Gillingham, J. Uesato, P.-S. Huang, R. Comanescu, F. Yang, A. See, S. Dathathri, R. Greig, C. Chen, D. Fritz, J. S. Elias, R. Green, S. Mokra, N. Fernando, B. Wu, R. Foley, S. Young, I. Gabriel, W. Isaac, J. Mellor, D. Hassabis, K. Kavukcuoglu, L. A. Hendricks, and G. Irving. Improving Alignment of Dialogue Agents via Targeted Human Judgements, 2022.

[117] N. Golmant, N. Vemuri, Z. Yao, V. Feinberg, A. Gholami, K. Rothauge, M. W. Mahoney, and J. E. Gonzalez. On the Computational Inefficiency of Large Batch Sizes for Stochastic Gradient Descent. *ArXiv*, abs/1811.12941, 2018.

[118] E. Gorbunov, F. Hanzely, and P. Richtárik. Local SGD: Unified Theory and New Efficient Methods, 2020.

[119] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, 2018.

[120] A. Griewank and A. Walther. Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Trans. Math. Softw.*, 26(1):19–45, mar 2000.

[121] D. Grishchenko, F. Iutzeler, J. Malick, and M.-R. Amini. Distributed Learning with Sparse Communications by Identification, 2020.

[122] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over Commodity Ethernet at Scale. *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[123] E. Guo, K. Wang, and Y. Zhang. Efficient Gradient Descent via Value Staleness Analysis for Heterogeneous Deep Learning Systems. In *2019 Seventh International Conference on Advanced Cloud and Big Data (CBD)*, pages 31–36, 2019.

[124] F. Haddadpour, M. M. Kamani, M. Mahdavi, and V. R. Cadambe. Local SGD with Periodic Averaging: Tighter Analysis and Adaptive Synchronization, 2020.

[125] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding, 2016.

[126] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both Weights and Connections for Efficient Neural Networks, 2015.

[127] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng. Deep Speech: Scaling up end-to-end speech recognition, 2014.

[128] T. Hansen. OpenAI 2727 Elo rating. 2025.

[129] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling, 2018.

[130] C. He, S. Li, J. So, X. Zeng, M. Zhang, H. Wang, X. Wang, P. Vepakomma, A. Singh, H. Qiu, X. Zhu, J. Wang, L. Shen, P. Zhao, Y. Kang, Y. Liu, R. Raskar, Q. Yang, M. Annavaram, and S. Avestimehr. FedML: A Research Library and Benchmark for Federated Machine Learning, 2020.

[131] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition, 2015.

[132] Q. Ho, J. Cipar, H. Cui, J. K. Kim, S. Lee, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. NIPS'13, page 1223–1231, Red Hook, NY, USA, 2013. Curran Associates Inc.

[133] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[134] T. Hoefler. From Large Language Models to Reasoning Language Models - Three Eras in The Age of Computation.

[135] E. Hoffer, I. Hubara, and D. Soudry. Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks. *ArXiv*, abs/1705.08741, 2017.

[136] J. E. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, and W. Chen. LoRA: Low-Rank Adaptation of Large Language Models. *ArXiv*, abs/2106.09685, 2021.

[137] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism, 2019.

[138] Z. Huo, Q. Yang, B. Gu, L. Carin, and H. Huang. Faster On-Device Training Using New Federated Momentum Algorithm. *ArXiv*, abs/2002.02090, 2020.

[139] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015.

[140] S. Jaghouar, J. M. Ong, and J. Hagemann. OpenDiLoCo: An Open-Source Framework for Globally Distributed Low-Communication Training. *ArXiv*, abs/2407.07852, 2024.

[141] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko. Priority-based Parameter Propagation for Distributed DNN Training, 2019.

[142] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, S. Shi, and X. Chu. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *ArXiv*, abs/1807.11205, 2018.

[143] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding, 2014.

[144] Z. Jia, M. A. Zaharia, and A. Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. *ArXiv*, abs/1807.05358, 2018.

[145] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. de Las Casas, E. B. Hanna, F. Bressand, G. Lengyel, G. Bour, G. Lample, L. R. Lavaud, L. Saulnier, M.-A. Lachaux, P. Stock, S. Subramanian, S. Yang, S. Antoniak, T. L. Scao, T. Gervet, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed. Mixtral of Experts. *ArXiv*, abs/2401.04088, 2024.

[146] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[147] P. H. Jin, Q. Yuan, F. Iandola, and K. Keutzer. How to scale distributed deep learning?, 2016.

[148] N. Jones. The AI revolution is running out of data. What can researchers do? *Nature*, 636 8042:290–292, 2024.

[149] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling Laws for Neural Language Models. *ArXiv*, abs/2001.08361, 2020.

[150] S. P. Karimireddy, S. Kale, M. Mohri, S. J. Reddi, S. U. Stich, and A. T. Suresh. SCAFFOLD: Stochastic Controlled Averaging for Federated Learning. In *International Conference on Machine Learning*, 2019.

[151] S. P. Karimireddy, Q. Rebjock, S. U. Stich, and M. Jaggi. Error Feedback Fixes SignSGD and other Gradient Compression Schemes, 2019.

[152] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima, 2017.

[153] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[154] Y. Ko, K. Choi, J. Seo, and S.-W. Kim. An In-Depth Analysis of Distributed Training of Deep Neural Networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 994–1003, 2021.

[155] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon. Federated Learning: Strategies for Improving Communication Efficiency, 2017.

[156] V. Kovalev, A. Kalinovsky, and S. Kovalev. Deep learning with Theano, Torch, Caffe, TensorFlow, and DeepLearning4J: Which one is the best in Speed and Accuracy? 2016.

[157] J. Kreps. Kafka : A Distributed Messaging System for Log Processing. 2011.

[158] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM*, 60(6):84–90, may 2017.

[159] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics (NRL)*, 52, 1955.

[160] S. K. Kumar. On weight initialization in deep neural networks. *ArXiv*, abs/1704.08863, 2017.

[161] F. Lai, Y. Dai, X. Zhu, and M. Chowdhury. FedScale: Benchmarking Model and System Performance of Federated Learning. *Proceedings of the First Workshop on Systems Challenges in Reliable and Secure Federated Learning*, 2021.

[162] M. Langer, Z. He, W. Rahayu, and Y. Xue. Distributed Training of Deep Learning Models: A Taxonomic Perspective. *IEEE Trans. Parallel Distrib. Syst.*, 31(12):2802–2818, dec 2020.

[163] Y. LeCun and Y. Bengio. *Convolutional Networks for Images, Speech, and Time Series*, page 255–258. MIT Press, Cambridge, MA, USA, 1998.

[164] Y. LeCun, C. Cortes, and C. Burges. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[165] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal Brain Damage. In *NIPS*, 1989.

[166] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension, 2019.

[167] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2020.

[168] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, Oct. 2014. USENIX Association.

[169] M. Li, D. G. Andersen, and A. Smola. Distributed Delayed Proximal Gradient Methods. 2013.

[170] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala. PyTorch Distributed: Experiences on Accelerating Data-Parallel Training. *Proc. VLDB Endow.*, 13(12):3005–3018, aug 2020.

[171] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith. Federated Learning: Challenges, Methods, and Future Directions. *IEEE Signal Processing Magazine*, 37(3):50–60, may 2020.

[172] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith. Federated Optimization in Heterogeneous Networks, 2020.

[173] X. Li, M. Jiang, X. Zhang, M. Kamp, and Q. Dou. FedBN: Federated Learning on Non-IID Features via Local Batch Normalization, 2021.

[174] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. NUMA-aware Algorithms: The case of Data Shuffling. In *Conference on Innovative Data Systems Research*, 2013.

[175] Z. Li, S. He, P. Chaturvedi, T.-H. Hoang, M. Ryu, E. A. Huerta, V. Kindratenko, J. D. Fuhrman, M. L. Giger, R. Chard, K. Kim, and R. K. Madduri. APPFLx: Providing Privacy-Preserving Cross-Silo Federated Learning as a Service. *2023 IEEE 19th International Conference on e-Science (e-Science)*, pages 1–4, 2023.

[176] Z. Li, S. He, Z. Yang, M. Ryu, K. Kim, and R. Madduri. Advances in APPFL: A Comprehensive and Extensible Federated Learning Framework. *ArXiv*, abs/2409.11585, 2024.

[177] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, and I. Stoica. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving.

[178] X. Lian, Y. Huang, Y. Li, and J. Liu. Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 2737–2745, Cambridge, MA, USA, 2015. MIT Press.

[179] X. Lian, Y. Huang, Y. Li, and J. Liu. Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 2737–2745, Cambridge, MA, USA, 2015. MIT Press.

[180] X. Lian, Y. Huang, Y. Li, and J. Liu. Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization, 2019.

[181] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu. Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent, 2017.

[182] X. Lian, W. Zhang, C. Zhang, and J. Liu. Asynchronous Decentralized Parallel Stochastic Gradient Descent, 2018.

[183] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello. Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets. *2018 IEEE International Conference on Big Data (Big Data)*, pages 438–447, 2018.

[184] X. Liang, S. Shen, E. Chen, J. Liu, Q. Liu, Y. Cheng, and Z. Pan. Accelerating local SGD for non-IID data using variance reduction. In *Frontiers of Computer Science*, 2022.

[185] T. Lin, L. Kong, S. U. Stich, and M. Jaggi. Extrapolation for Large-Batch Training in Deep Learning. *ArXiv*, abs/2006.05720, 2020.

[186] T. Lin, S. U. Stich, K. K. Patel, and M. Jaggi. Don't Use Large Mini-Batches, Use Local SGD, 2020.

[187] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training, 2020.

[188] P. Lindstrom. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20:2674–2683, 2014.

[189] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45:503–528, 1989.

[190] Y. Liu, X. Lu, Q. Wu, C. Xie, and X. Tang. TCP is CPU intensive: Fixing performance problems in an IP-based storage system. In *Proceedings of the 2005 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 75–84. ACM, 2005.

[191] L. Long, R. Wang, R. Xiao, J. Zhao, X. Ding, G. Chen, and H. Wang. On LLMs-Driven Synthetic Data Generation, Curation, and Evaluation: A Survey. In *Annual Meeting of the Association for Computational Linguistics*, 2024.

[192] I. Loshchilov and F. Hutter. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*, 2017.

[193] S. McCandlish, J. Kaplan, D. Amodei, and O. D. Team. An Empirical Model of Large-Batch Training, 2018.

[194] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data.

[195] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine Learning in Apache Spark, 2015.

[196] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed Precision Training. *ArXiv*, abs/1710.03740, 2017.

[197] I. Mitliagkas, C. Zhang, S. Hadjis, and C. Ré. Asynchrony begets Momentum, with an Application to Deep Learning, 2016.

[198] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning Convolutional Neural Networks for Resource Efficient Inference, 2017.

[199] G. E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE Solid-State Circuits Society Newsletter*, 86:82–85, 1998.

[200] V. Nair and G. E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

[201] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[202] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. A. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. A. Zaharia. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[203] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust Stochastic Approximation Approach to Stochastic Programming. *SIAM J. on Optimization*, 19(4):1574–1609, jan 2009.

[204] Y. Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$). *Proceedings of the USSR Academy of Sciences*, 269:543–547, 1983.

[205] F. Niu, B. Recht, C. Re, and S. J. Wright. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent, 2011.

[206] N. P. D. P. Norman P. Jouppi, Cliff Young et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. 2017.

[207] OpenAI. GPT-4 Technical Report, 2024.

[208] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.

[209] J. H. Park, G. Yun, C. M. Yi, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y. ri Choi. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 307–321. USENIX Association, July 2020.

[210] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library, 2019.

[211] P. Patarasuk and X. Yuan. Bandwidth Optimal All-Reduce Algorithms for Clusters of Workstations. *J. Parallel Distrib. Comput.*, 69(2):117–124, feb 2009.

[212] X. Peng, Q. Bai, X. Xia, Z. Huang, K. Saenko, and B. Wang. Moment Matching for Multi-Source Domain Adaptation, 2019.

[213] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[214] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai. Privacy-Preserving Deep Learning via Additively Homomorphic Encryption. *IEEE Transactions on Information Forensics and Security*, 13:1333–1345, 2018.

[215] J. Qiu, B. Peng, R. Teja, S. Tyagi, C. Widanage, and J. Koskey. Real-Time Anomaly Detection from Edge to HPC-Cloud. 2018.

[216] R. Rabenseifner. Optimization of collective reduction operations. In *International Conference on Conceptual Structures*, 2004.

[217] A. Radford and K. Narasimhan. Improving Language Understanding by Generative Pre-Training. 2018.

[218] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language Models are Unsupervised Multitask Learners. 2019.

[219] R. Rafailov, A. Sharma, E. Mitchell, S. Ermon, C. D. Manning, and C. Finn. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. *ArXiv*, abs/2305.18290, 2023.

[220] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, 2020.

[221] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models, 2020.

[222] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen. Hierarchical Text-Conditional Image Generation with CLIP Latents. *ArXiv*, abs/2204.06125, 2022.

[223] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever. Zero-Shot Text-to-Image Generation, 2021.

[224] G. Raposo, P. Tomás, and N. Roma. PositNN: Training Deep Neural Networks with Mixed Low-Precision Posit. *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7908–7912, 2021.

[225] S. Rashidi, M. Denton, S. Sridharan, S. Srinivasan, A. Suresh, J. Nie, and T. Krishna. Enabling Compute-Communication Overlap in Distributed Deep Learning Training Platforms. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 540–553, 2021.

[226] S. Reddi, Z. Charles, M. Zaheer, Z. Garrett, K. Rush, J. Konečný, S. Kumar, and H. B. McMahan. Adaptive Federated Optimization, 2021.

[227] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He. ZeRO-Offload: Democratizing Billion-Scale Model Training. *ArXiv*, abs/2101.06840, 2021.

[228] C. Renggli, S. Ashkboos, M. Aghagolzadeh, D. Alistarh, and T. Hoefler. SparCML: High-Performance Sparse Communication for Machine Learning, 2019.

[229] G. Research. PaLM: Scaling Language Modeling with Pathways, 2022.

[230] R. O. Rogers and D. B. Skillicorn. Using the BSP Cost Model to Optimize Parallel Neural Network Training. *Future Gener. Comput. Syst.*, 14(5–6):409–424, dec 1998.

[231] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-Resolution Image Synthesis with Latent Diffusion Models, 2022.

[232] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[233] H. R. Roth, Y. Cheng, Y. Wen, I. Yang, Z. Xu, Y.-T. Hsieh, K. Kersten, A. E. Harouni, C. Zhao, K. Lu, Z. Zhang, W. Li, A. Myronenko, D. Yang, S. B. Yang, N. Rieke, A. Quraini, C. Chen, D. Xu, N. Ma, P. Dogra, M. G. Flores, and A. Feng. NVIDIA FLARE: Federated Learning from Simulation to Real-World. *ArXiv*, abs/2210.13291, 2022.

[234] D. Rothchild, A. Panda, E. Ullah, N. Ivkin, I. Stoica, V. Braverman, J. Gonzalez, and R. Arora. FetchSGD: Communication-Efficient Federated Learning with Sketching. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[235] S. Ruder. An Overview of Gradient Descent Optimization Algorithms, 2016.

[236] T. Ryffel, A. Trask, M. Dahl, B. Wagner, J. Mancuso, D. Rueckert, and J. Passerat-Palmbach. A generic framework for privacy preserving deep learning, 2018.

[237] M. Ryu, Y. Kim, K. Kim, and R. K. Madduri. APPFL: Open-Source Software Framework for Privacy-Preserving Federated Learning. *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1074–1083, 2022.

[238] P. Sanders, J. Speck, and J. L. Träff. Two-Tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan. *Parallel Comput.*, 35(12):581–594, dec 2009.

[239] F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek. Robust and Communication-Efficient Federated Learning From Non-i.i.d. Data. *IEEE Transactions on Neural Networks and Learning Systems*, 31(9):3400–3413, 2020.

[240] F. Seide and A. Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 2135, New York, NY, USA, 2016. Association for Computing Machinery.

[241] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit Stochastic Gradient Descent and its application to Data-Parallel Distributed Training of Speech DNNs. In *Interspeech*, 2014.

[242] A. Sergeev and M. D. Balso. Horovod: Fast and easy distributed deep learning in TensorFlow, 2018.

[243] J. Sevilla, L. Heim, A. C. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos. Compute Trends Across Three Eras of Machine Learning. *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2022.

[244] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl. Measuring the Effects of Data Parallelism on Neural Network Training, 2019.

[245] N. M. Shazeer and M. Stern. Adafactor: Adaptive Learning Rates with Sublinear Memory Cost. *ArXiv*, abs/1804.04235, 2018.

[246] H. Shi, Y. Zhao, B. Zhang, K. Yoshigoe, and F. Chang. Effective Parallel Computing via a Free Stale Synchronous Parallel Strategy. *IEEE Access*, 7:118764–118775, 2019.

[247] S. Shi, X. Chu, K. C. Cheung, and S. See. Understanding Top-k Sparsification in Distributed Deep Learning, 2019.

[248] S. Shi, X. Chu, and B. Li. MG-WFBP: Efficient Data Communication for Distributed Synchronous SGD Algorithms, 2018.

[249] S. Shi, Q. Wang, K. Zhao, Z. Tang, Y. Wang, X. Huang, and X. Chu. A Distributed Synchronous SGD Algorithm with Global Top-k Sparsification for Low Bandwidth Networks, 2019.

[250] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *ArXiv*, abs/1909.08053, 2019.

[251] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[252] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[253] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *ArXiv*, abs/1712.01815, 2017.

[254] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2015.

[255] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Y. Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model, 2022.

[256] S. L. Smith and Q. V. Le. A Bayesian Perspective on Generalization and Stochastic Gradient Descent. *ArXiv*, abs/1710.06451, 2017.

[257] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen. AccPar: Tensor Partitioning for Heterogeneous Deep Learning Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 342–355, 2020.

[258] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen. HyPar: Towards Hybrid Parallelism for Deep Learning Accelerator Array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 56–68, 2019.

[259] S. Srinivas and R. V. Babu. Data-free parameter pruning for Deep Neural Networks, 2015.

[260] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, jan 2014.

[261] T. L. Sterling, M. Anderson, and M. Brodowicz. High Performance Computing: Modern Systems and Practices. 2017.

[262] S. U. Stich. Local SGD Converges Fast and Communicates Little, 2019.

[263] S. U. Stich, J.-B. Cordonnier, and M. Jaggi. Sparsified SGD with Memory. *ArXiv*, abs/1809.07599, 2018.

[264] N. Strom. Scalable distributed DNN training using commodity GPU cloud computing. In *Interspeech*, 2015.

[265] Z. Tang, S. Shi, X. Chu, W. Wang, and B. Li. Communication-Efficient Distributed Deep Learning: A Comprehensive Survey. *ArXiv*, abs/2003.06307, 2020.

[266] G. G. Team. Gemini: A Family of Highly Capable Multimodal Models, 2024.

[267] G. P. Team. PaLM 2 Technical Report, 2023.

[268] M. L. Team. The LLaMA 3 Herd of Models, 2024.

[269] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, feb 2005.

[270] Theano Development Team. Theano: A Python Framework for Fast Computation of Mathematical Expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[271] R. Thibaux and M. I. Jordan. Hierarchical Beta Processes and the Indian Buffet Process. In M. Meila and X. Shen, editors, *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, volume 2 of *Proceedings of Machine Learning Research*, pages 564–571, San Juan, Puerto Rico, 21–24 Mar 2007. PMLR.

[272] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy. Storm@Twitter. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.

[273] Y. Tsuzuku, H. Imachi, and T. Akiba. Variance-based Gradient Compression for Efficient Distributed Deep Learning, 2018.

[274] S. Tyagi and P. Sharma. Taming Resource Heterogeneity In Distributed ML Training With Dynamic Batching. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 188–194, 2020.

[275] S. Tyagi and P. Sharma. Scavenger: A Cloud Service for Optimizing Cost and Performance of ML Training, 2023.

[276] S. Tyagi and P. Sharma. OmniLearn: A Framework for Distributed Deep Learning over Heterogeneous Clusters . *IEEE Transactions on Parallel & Distributed Systems*, (01):1–15, Mar. 2025.

[277] S. Tyagi and M. Swany. ScaDLES: Scalable Deep Learning over Streaming data at the Edge. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 2113–2122, Los Alamitos, CA, USA, dec 2022. IEEE Computer Society.

[278] S. Tyagi and M. Swany. Accelerating Distributed ML Training via Selective Synchronization. *2023 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, 2023.

[279] S. Tyagi and M. Swany. Flexible Communication for Optimal Distributed Learning over Unpredictable Networks. *2023 IEEE International Conference on Big Data (BigData)*, pages 925–935, 2023.

[280] S. Tyagi and M. Swany. GraVAC: Adaptive Compression for Communication-Efficient Distributed DL Training. *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pages 319–329, 2023.

[281] Y. Ueno and R. Yokota. Exhaustive Study of Hierarchical AllReduce Patterns for Large Messages Between GPUs. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 430–439, 2019.

[282] J. van Leeuwen. On the Construction of Huffman Trees. In *International Colloquium on Automata, Languages and Programming*, 1976.

[283] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need, 2017.

[284] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer. A Survey on Distributed Machine Learning. *ACM Comput. Surv.*, 53(2), mar 2020.

[285] T. Vogels, S. P. Karimireddy, and M. Jaggi. PowerGossip: Practical Low-Rank Communication Compression in Decentralized Deep Learning. *ArXiv*, abs/2008.01425, 2020.

[286] T. Vogels, S. P. Karimireddy, and M. Jaggi. PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization, 2020.

[287] B. Wang, Q. Xu, Z. Bian, and Y. You. Tesseract: Parallelize the Tensor Parallelism Efficiently. *Proceedings of the 51st International Conference on Parallel Processing*, 2021.

[288] E. Wang, Q. Zhang, S. Bo, G. Zhang, X. Lu, Q. Wu, and Y. Wang. Intel Math Kernel Library. 2014.

[289] G. Wang, H. Qin, S. A. Jacobs, C. Holmes, S. Rajbhandari, O. Ruwase, F. Yan, L. Yang, and Y. He. ZeRO++: Extremely Efficient Collective Communication for Giant Model Training. *ArXiv*, abs/2306.10209, 2023.

[290] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda. GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2595–2605, 2014.

[291] H. Wang, S. Sievert, Z. Charles, S. Liu, S. Wright, and D. Papailiopoulos. ATOMO: Communication-efficient learning via Atomic Sparsification, 2018.

[292] J. Wang and G. Joshi. Adaptive Communication Strategies to Achieve the Best Error-Runtime Trade-off in Local-Update SGD, 2019.

[293] J. Wang, Q. Liu, H. Liang, G. Joshi, and H. V. Poor. Tackling the Objective Inconsistency Problem in Heterogeneous Federated Optimization, 2020.

[294] Q. Wang, Y. Ma, K. Zhao, and Y. jie Tian. A Comprehensive Survey of Loss Functions in Machine Learning. *Annals of Data Science*, 9:187 – 212, 2020.

[295] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan. Adaptive Federated Learning in Resource Constrained Edge Computing Systems. *IEEE Journal on Selected Areas in Communications*, 37(6):1205–1221, 2019.

[296] 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, 2017.

[297] Y. Wen, K. Luk, M. Gazeau, G. Zhang, H. Chan, and J. Ba. An Empirical Study of Large-Batch Stochastic Gradient Descent with Structured Covariance Noise. *arXiv: Learning*, 2019.

[298] C. Widanage, J. Li, S. Tyagi, R. Teja, B. Peng, S. Kamburugamuve, D. Baum, D. Smith, J. Qiu, and J. Koskey. Anomaly Detection over Streaming Data: Indy500 Case Study. *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 9–16, 2019.

[299] B. Woodworth, K. K. Patel, and N. Srebro. Minibatch vs Local SGD for Heterogeneous Distributed Learning, 2022.

[300] B. Workshop, :, T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé, J. Tow, A. M. Rush, S. Biderman, A. Webson, P. S. Ammanamanchi, T. Wang, B. Sagot, N. Muennighoff, A. V. del Moral, O. Ruwase, R. Bawden, S. Bekman, A. McMillan-Major, I. Beltagy, and et al. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model, 2023.

[301] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A New Platform for Distributed Machine Learning on Big Data, 2015.

[302] A. Xu, Z. Huo, and H. Huang. On the Acceleration of Deep Learning Model Parallelism With Staleness. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2085–2094, 2020.

[303] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis. GRACE: A compressed communication framework for distributed machine learning. In *Proc. of 41st IEEE Int. Conf. Distributed Computing Systems (ICDCS)*, 2021.

[304] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. H. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis. Compressed Communication for Distributed Deep Learning: Survey and Quantitative Evaluation. 2020.

[305] Y. Xue, D. Klabjan, and Y. Luo. Aggregation Delayed Federated Learning. *2022 IEEE International Conference on Big Data (Big Data)*, pages 85–94, 2021.

[306] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le. XLNet: Generalized Autoregressive Pre-training for Language Understanding, 2020.

[307] Z. Yao, A. Gholami, Q. Lei, K. Keutzer, and M. W. Mahoney. Hessian-based Analysis of Large Batch Training and Robustness to Adversaries. In *Neural Information Processing Systems*, 2018.

[308] Y. You, I. Gitman, and B. Ginsburg. Large Batch Training of Convolutional Networks. *arXiv: Computer Vision and Pattern Recognition*, 2017.

[309] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer. ImageNet Training in Minutes. *Proceedings of the 47th International Conference on Parallel Processing*, 2017.

[310] C.-C. Yu and B.-D. Liu. A backpropagation algorithm with adaptive learning rate and momentum coefficient. *Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No.02CH37290)*, 2:1218–1223 vol.2, 2002.

[311] M. Yurochkin, M. Agarwal, S. Ghosh, K. Greenewald, T. N. Hoang, and Y. Khazaeni. Bayesian Nonparametric Federated Learning of Neural Networks, 2019.

[312] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 59(11):56–65, oct 2016.

[313] M. A. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Symposium on Networked Systems Design and Implementation*, 2012.

[314] M. Zaheer, S. J. Reddi, D. Sachan, S. Kale, and S. Kumar. Adaptive Methods for Nonconvex Optimization. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 9815–9825, Red Hook, NY, USA, 2018. Curran Associates Inc.

[315] M. D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *ArXiv*, abs/1212.5701, 2012.

[316] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *USENIX Annual Technical Conference*, 2017.

[317] S. Zhang, A. Choromanska, and Y. LeCun. Deep learning with Elastic Averaging SGD, 2015.

[318] W. Zhang, S. Gupta, X. Lian, and J. Liu. Staleness-Aware Async-SGD for Distributed Deep Learning. *ArXiv*, abs/1511.05950, 2015.

[319] Y. Zhang, S. Sun, M. Galley, Y.-C. Chen, C. Brockett, X. Gao, J. Gao, J. Liu, and B. Dolan. DialoGPT: Large-Scale Generative Pre-training for Conversational Response Generation, 2020.

[320] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin. Is Network the Bottleneck of Distributed Training? In *Proceedings of the Workshop on Network Meets AI and ML*, NetAI '20, page 8–13, New York, NY, USA, 2020. Association for Computing Machinery.

[321] K. Zhao, S. Di, M. Dmitriev, T. Tonellot, Z. Chen, and F. Cappello. Optimizing Error-Bounded Lossy Compression for Scientific Data by Dynamic Spline Interpolation. *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1643–1654, 2021.

[322] S. Zhao, F. Li, X. Chen, T. Shen, L. Chen, S. Wang, N. Zhang, C. Li, and H. Cui. NASPipe: High Performance and Reproducible Pipeline Parallel Supernet Training via Causal Synchronous Parallelism. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 374–387, New York, NY, USA, 2022. Association for Computing Machinery.

[323] X. Zhao, A. An, J. Liu, and B. X. Chen. Dynamic Stale Synchronous Parallel Distributed Training for Deep Learning, 2019.

[324] Y. Zhao, A. Gu, R. Varma, L. Luo, C. chin Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, A. Desmaison, C. Balioglu, B. Nguyen, G. Chauhan, Y. Hao, and S. Li. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.*, 16:3848–3860, 2023.

[325] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra. Federated Learning with Non-IID Data. 2018.

[326] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica. Alpa: Automating inter- and intra-Operator Parallelism for Distributed Deep Learning, 2022.

[327] S. Zheng, Z. Huang, and J. T. Kwok. Communication-Efficient Distributed Blockwise Momentum SGD with Error-Feedback, 2019.

[328] Y. Zhuang, H. Zhao, L. Zheng, Z. Li, E. P. Xing, Q. Ho, J. E. Gonzalez, I. Stoica, and H. Zhang. On Optimizing the Communication of Model Parallelism, 2022.

[329] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized Stochastic Gradient Descent. In *NIPS*, 2010.