A Survey on Distributed Machine Learning

Verbraeken, Joost; Wolting, Matthijs; Katzy, Jonathan ; Kloppenburg, Jeroen; Verbelen, Tim; Rellermeyer, Jan S.

DOI
10.1145/3377454

Publication date
2020

Document Version
Final published version

Published in
ACM Computing Surveys (CSUR)

A Survey on Distributed Machine Learning

JOOST VERBRAEKEN, MATTHIJS WOLTING, JONATHAN KATZY, and
JEROEN KLOPPENBURG, Delft University of Technology, Netherlands
TIM VERBELEN, imec - Ghent University, Belgium
JAN S. RELLERMEYER, Delft University of Technology, Netherlands

The demand for artificial intelligence has grown significantly over the past decade, and this growth has been fueled by advances in machine learning techniques and the ability to leverage hardware acceleration. However, to increase the quality of predictions and render machine learning solutions feasible for more complex applications, a substantial amount of training data is required. Although small machine learning models can be trained with modest amounts of data, the input for training larger models such as neural networks grows exponentially with the number of parameters. Since the demand for processing training data has outpaced the increase in computation power of computing machinery, there is a need for distributing the machine learning workload across multiple machines, and turning the centralized into a distributed system. These distributed systems present new challenges: first and foremost, the efficient parallelization of the training process and the creation of a coherent model. This article provides an extensive overview of the current state-of-the-art in the field by outlining the challenges and opportunities of distributed machine learning over conventional (centralized) machine learning, discussing the techniques used for distributed machine learning, and providing an overview of the systems that are available.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Computing methodologies** → **Machine learning**; • **Computer systems organization** → **Distributed architectures**;

Additional Key Words and Phrases: Distributed machine learning, distributed systems

ACM Reference format:
Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. 2020. A Survey on Distributed Machine Learning. *ACM Comput. Surv.* 53, 2, Article 30 (March 2020), 33 pages.
https://doi.org/10.1145/3377454

1 INTRODUCTION

The rapid development of new technologies in recent years has led to an unprecedented growth of data collection. Machine Learning (ML) algorithms are increasingly being used to analyze datasets and build decision-making systems for which an algorithmic solution is not feasible due

Authors' addresses: J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, and J. S. Rellermeyer, Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Van Mourik Broekmanweg 6, 2628XE Delft, Netherlands; emails: J.Verbraeken@student.tudelft.nl, matthijswolting@gmail.com, {J.B.Katzy, J.Kloppenburg}@student.tudelft.nl, j.s.rellermeyer@tudelft.nl; T. Verbelen, Ghent University, IDLab, Department of Information Technology, Technologiepark 126, 9052 Ghent, Belgium; email: tim.verbelen@ugent.be.

0360-0300/2020/03-ART30 $15.00
https://doi.org/10.1145/3377454

to the complexity of the problem. Examples include controlling self-driving cars [23], recognizing speech [8], or predicting consumer behavior [82].

In some cases, the long runtime of training the models steers solution designers towards using distributed systems for an increase of parallelization and total amount of I/O bandwidth, as the training data required for sophisticated applications can easily be in the order of terabytes [29]. In other cases, a centralized solution is not even an option when data are inherently distributed or too big to store on single machines. Examples include transaction processing in larger enterprises on data that are stored in different locations [19] or astronomical data that are too large to move and centralize [124].

To make these types of datasets accessible as training data for machine learning problems, algorithms have to be chosen and implemented that enable parallel computation, data distribution, and resilience to failures. A rich and diverse ecosystem of research has been conducted in this field, which we categorize and discuss in this article. In contrast to prior surveys on distributed machine learning [119, 123] or related fields [87, 121, 122, 143, 152, 170], we apply a wholistic view to the problem and discuss the practical aspects of state-of-the-art machine learning from a distributed systems angle.

Section 2 provides an in-depth discussion of the system challenges of machine learning and how ideas from High Performance Computing (HPC) have been adopted for acceleration and increased scalability. Section 3 describes a reference architecture for distributed machine learning covering the entire stack from algorithms to the network communication patterns that can be employed to exchange state between individual nodes. Section 4 presents the ecosystem of the most widely used systems and libraries as well as their underlying designs. Finally, Section 5 discusses the main challenges of distributed machine learning.

2 MACHINE LEARNING—A HIGH-PERFORMANCE COMPUTING CHALLENGE?

Recent years have seen a proliferation of machine learning technology in increasingly complex applications. While various competing approaches and algorithms have emerged, the data representations used are strikingly similar in structure. The majority of computation in machine learning workloads amounts to basic transformations on vectors, matrices, or tensors—well-known problems from linear algebra. The need to optimize such operations has been a highly active area of research in the high-performance computing community for decades. As a result, some techniques and libraries from the HPC community (e.g., BLAS [89] or MPI [62]) have been successfully adopted and integrated into systems by the machine learning community. At the same time, the HPC community has identified machine learning to be an emerging high-value workload and has started to apply HPC methodology to them. Coates et al. [38] were able to train a 1B parameter network on their Commodity Off-The-Shelf High Performance Computing (COTS HPC) system in just three days. You et al. [165] optimized the training of a neural network on Intel's Knights Landing, a chip designed for HPC applications. Kurth et al. [84] demonstrated how deep learning problems like extracting weather patterns can be optimized and scaled efficiently on large parallel HPC systems. Yan et al. [162] have addressed the challenge of scheduling deep neural network applications on cloud computing infrastructure by modeling the workload demand with techniques like lightweight profiling, which are borrowed from HPC. Li et al. [91] investigated the resilience characteristics of deep neural networks with regard to hardware errors when running on accelerators, which are frequently deployed in major HPC systems.

Like for other large-scale computational challenges, there are two fundamentally different and complementary ways of accelerating workloads: adding more resources to a single machine (vertical scaling or scaling up) and adding more nodes to the system (horizontal scaling or scaling out).

2.1 Scaling Up

Among the scale-up solutions, adding programmable GPUs is the most common method and various systematic efforts have shown the benefits of doing so [18, 78, 125]. GPUs feature a high number of hardware threads. For example, the Nvidia Titan V and Nvidia Tesla V100 have a total of 5,120 cores, which makes them approximately 47× faster for deep learning than a regular server CPU (namely an Intel Xeon E5-2690v4) [107]. Originally the applications of GPUs for machine learning were limited because GPUs used a pure SIMD (Single Instruction, Multiple Data) [51] model that did not allow the cores to execute a different branch of the code; all threads had to perform the exact same program. Over the years GPUs have shifted to more flexible architectures where the overhead of branch divergence is reduced, but diverging branches is still inefficient [66]. The proliferation of GPGPUs (General-Purpose GPUs, i.e., GPUs that can execute arbitrary code) has lead the vendors to design custom products that can be added to conventional machines as accelerators and no longer fulfill any role in the graphics subsystem of the machine. For example, the Nvidia Tesla GPU series is meant for highly parallel computing and designed for deployment in supercomputers and clusters. When a sufficient degree of parallelism is offered by the workload, GPUs can significantly accelerate machine learning algorithms. For example, Meuth [100] reported a speed-up up to 200× over conventional CPUs for an image recognition algorithm using a Pretrained Multilayer Perceptron (MLP).

An alternative to generic GPUs for acceleration is the use of Application Specific Integrated Circuits (ASICs), which implement specialized functions through a highly optimized design. In recent times, the demand for such chips has risen significantly [99]. When applied to, e.g., Bitcoin mining, ASICs have a significant competitive advantage over GPUs and CPUs due to their high performance and power efficiency [144]. Since matrix multiplications play a prominent role in many machine learning algorithms, these workloads are highly amenable to acceleration through ASICs. Google applied this concept in their Tensor Processing Unit (TPU) [128], which, as the name suggests, is an ASIC that specializes in calculations on tensors (n-dimensional arrays), and is designed to accelerate their Tensorflow [1, 2] framework, a popular building block for machine learning models. The most important component of the TPU is its Matrix Multiply unit based on a systolic array. TPUs use a MIMD (Multiple Instructions, Multiple Data) [51] architecture that, unlike GPUs, allows them to execute diverging branches efficiently. TPUs are attached to the server system through the PCI Express bus. This provides them with a direct connection with the CPU, which allows for a high aggregated bandwidth of 63 GB/s (PCI-e5x16). Multiple TPUs can be used in a data center, and the individual units can collaborate in a distributed setting. The benefit of the TPU over regular CPU/GPU setups is not only its increased processing power but also its power efficiency, which is important in large-scale applications due to the cost of energy and the limited availability in large-scale data centers. When running benchmarks, Jouppi et al. [80] found that the performance per watt of a TPU can approach 200× that of a traditional system. Further benchmarking by Sato et al. [128] indicated that the total processing power of a TPU or GPU can be up to 70× higher than a CPU for a typical neural network, with performance improvements varying from 3.5×–71×, depending on the task at hand.

Chen et al. [32] developed DianNao, a hardware accelerator for large-scale neural networks with a small area footprint. Their design introduces a Neuro-Functional Unit (NFU) in a pipeline that multiplies all inputs, adds the results, and, in a staggered manner after all additions have been performed, optionally applies an activation function like a sigmoid function. The experimental evaluation using the different layers of several large neural network structures [48, 70, 90, 132, 133] shows a performance speedup of three orders of magnitude and an energy reduction of more than 20× compared to using a general-purpose 128-bit 2 GHz SIMD CPU.

Hinton et al. [70] address the challenge that accessing the weights of neurons from DRAM is a costly operation and can dominate the energy profile of processing. Leveraging a *deep compression* technique, they are able to put the weights into SRAM and accelerate the resulting sparse matrix-vector multiplications through efficient weight sharing. The result is a 2.9× higher throughput and a 19× improved energy efficiency compared to DianNao.

Even general-purpose CPUs have increased the availability and width of vector instructions in recent product generations to accelerate the processing of computationally intensive problems like machine learning algorithms. These instructions are vector instructions, part of the AVX-512 family [126] with enhanced word-variable precision and support for single precision floating-point operations. In addition to the mainstream players, there are more specialized designs available such as the Epiphany [110]. This special-purpose CPU is designed with a MIMD architecture that uses an array of processors, each of which accessing the same memory, to speed up execution of floating-point operations. This is faster than giving every processor its own memory, because communicating between processors is expensive. The newest chip of the major manufacturer Adapteva is the Epiphany V, which contains 1,024 cores on a single chip [109]. Although Adapteva has not published power consumption specifications of the Epiphany V yet, it has released numbers suggesting a power usage of only 2 Watts [4].

2.2 Scaling Out

While there are many different strategies to increase the processing power of a single machine for large-scale machine learning, there are reasons to prefer a scale-out design or combine the two approaches, as often seen in HPC. The first reason is the generally lower equipment cost, both in terms of initial investment and maintenance. The second reason is the resilience against failures because, when a single processor fails within an HPC application, the system can still continue operating by initiating a partial recovery (e.g., based on communication-driven checkpointing [46] or partial re-computation [168]). The third reason is the increase in aggregate I/O bandwidth compared to a single machine [49]. Training ML models is a highly data-intensive task, and the ingestion of data can become a serious performance bottleneck [67]. Since every node has a dedicated I/O subsystem, scaling out is an effective technique for reducing the impact of I/O on the workload performance by effectively parallelizing the reads and writes over multiple machines. A major challenge of scaling out is that not all ML algorithms lend themselves to a distributed computing model, which can thus only be used for algorithms that can achieve a high degree of parallelism.

2.3 Discussion

The lines between traditional supercomputers, grids, and the cloud are increasingly getting blurred when it comes to the best execution environment for demanding workloads like machine learning. For instance, GPUs and accelerators are now more common in major cloud datacenters [134, 135]. As a result, parallelization of the machine learning workload has become paramount to achieving acceptable performance at large scale. When transitioning from a centralized solution to a distributed system, however, the typical challenges of distributed computing in the form of performance, scalability, failure resilience, or security apply [40]. The following section presents a systematic discussion of the different aspects of distributed machine learning and develops a reference architecture by which all existing systems can be categorized.

3 A REFERENCE ARCHITECTURE FOR DISTRIBUTED MACHINE LEARNING

Designing a generic system that enables an efficient distribution of regular machine learning is challenging, since every algorithm has a distinct communication pattern [78, 105, 127, 145, 149,

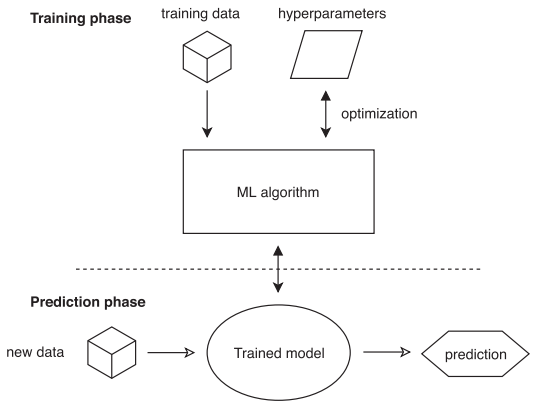

Fig. 1. General overview of machine learning. During the training phase an ML model is optimized using training data and by tuning hyper parameters. Then the trained model is deployed to provide predictions for new data fed into the system.

151]. Despite various different concepts and implementations for distributed machine learning, we have identified a common architectural framework that covers the entire design space. Every section discusses a particular area where designers of machine learning solutions need to make a decision.

In general, the problem of machine learning can be separated into the training and the prediction phase (Figure 1).

The *Training phase* involves training a machine learning model by feeding it a large body of training data and updating it using an ML algorithm. An overview of applicable and commonly used algorithms is given in Section 3.1. Aside from choosing a suitable algorithm for a given problem, we also need to find an optimal set of hyperparameters for the chosen algorithm, which is described in Section 3.2. The final outcome of the training phase is a *Trained Model*, which can then be deployed. The *Prediction phase* is used for deploying the trained model in practice. The trained model accepts new data as input and produces a prediction as output. While the training phase of the model is typically computationally intensive and requires the availability of large datasets, the inference can be performed with less computing power.

The training phase and prediction phase are not mutually exclusive. Incremental learning combines the training phase and inference phase and continuously trains the model by using new data from the prediction phase.

When it comes to distribution, there are two fundamentally different ways of partitioning the problem across all machines: parallelizing the data or the model [119] (Figure 2). These two methods can also be applied simultaneously [161].

In the *Data-Parallel* approach, the data are partitioned as many times as there are worker nodes in the system and all worker nodes subsequently apply the same algorithm to different datasets. The same model is available to all worker nodes (either through centralization or through replication) so a single coherent output emerges naturally. The technique can be used with every ML algorithm with an independent and identical distribution (i.i.d.) assumption over the data samples (i.e., most ML algorithms [161]). In the *Model-Parallel* approach, exact copies of the entire datasets are processed by the worker nodes that operate on different parts of the model. The model is therefore the aggregate of all model parts. The model-parallel approach cannot automatically be applied to every machine learning algorithm, because the model parameters generally cannot be split up.

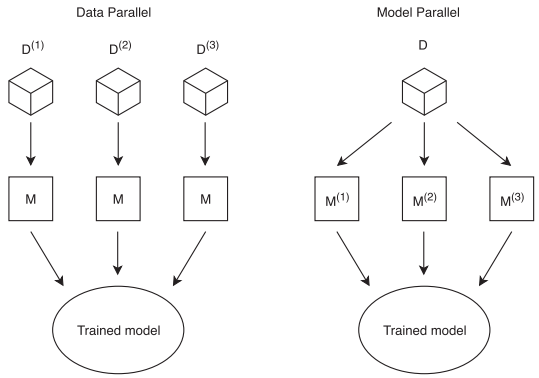

Fig. 2. Parallelism in distributed machine learning. Data parallelism trains multiple instances of the same model on different subsets of the training dataset, while model parallelism distributes parallel paths of a single model to multiple nodes.

One option is to train different instances of the same or similar model, and aggregate the outputs of all trained models using methodologies like ensembling (Section 3.3).

The final architectural decision is the *topology* of the distributed machine learning system. The different nodes that form the distributed system need to be connected through a specific architectural pattern to fulfill a common task. However, the choice of pattern has implications on the role that a node can play, the degree of communication between nodes, and the failure resilience of the whole deployment. A discussion of commonly used topologies is presented in Section 3.4.

In practice, the three layers of architecture (machine learning, parallelism, topology) are not independent. The combining factor is their impact on the amount of communication required to train the model, which is discussed in Section 3.5.

3.1 Machine Learning Algorithms

ML algorithms learn to make decisions or predictions based on data. We categorize current ML algorithms based on the following three characteristics:

- **Feedback**—the type of feedback that is given to the algorithm while learning.
- **Purpose**—the desired end result of the algorithm.
- **Method**—the nature of model evolution that occurs when given feedback.

3.1.1 Feedback. To train an algorithm, it requires feedback so it can gradually improve the quality of the model. There are several different types of feedback [164]:

- **Supervised learning** uses training data that consist of input objects (usually vectors) and the corresponding desired output values. Supervised learning algorithms attempt to find a function that maps the input data to the desired output. Then, this function can be applied to new input data to predict the output. One of the goals is to minimize both the bias and variance error of the predicted results. The bias error is caused by simplifying assumptions made by the learning algorithm to facilitate learning the target function. However, methods with high bias have lower predictive performance on problems that do not fully satisfy the assumptions. For example, a linear model will not be able to give accurate predictions if the underlying data have a non-linear behavior. The variance captures how much the results of the ML algorithm change for a different training set. A high variance means that the algorithm is modeling the specifics of the training data without finding the underlying

(hidden) mapping between the inputs and the outputs. Unfortunately, eliminating both the bias and the variance is typically impossible, a phenomenon known as the bias-variance trade-off [54]. The more complex the model, the more training data are required to train the algorithm to gain an accurate prediction from the model. For example, when the dimensionality of the data is high, the output may depend on a convoluted combination of input factors, which requires a high number of data samples to detect the relations between these dimensions.

- **Unsupervised learning** uses training data that consist of input objects (usually vectors) without output values. Unsupervised learning algorithms aim at finding a function that describes the structure of the data and group the unsorted input data. Because the input data are unlabeled, they lack a clear output accuracy metric. The most common use case of unsupervised learning is to cluster data together based on similarities and hidden patterns. Unsupervised learning is also used for problems like dimensionality reduction where the key features of data are extracted. In this case, the feedback is generated using a similarity metric.
- **Semi-supervised learning** uses a (generally small) amount of labeled data, supplemented by a comparatively large amount of unlabeled data. Clustering can be used to extrapolate known labels onto unlabeled data points. This is done under the assumption that similar data points share the same label.
- **Reinforcement learning** is used to train an agent that has to take actions in an environment based on its observations. Feedback relies on a reward or cost function that evaluates the states of the system. The biggest challenge here is the credit assignment problem, or how to determine which actions actually lead to higher reward in the long run. Bagnell and Ng [13] showed that a local reward system is beneficial for the scalability of the learning problem, since global schemes require samples that scale roughly linearly with the number of participating nodes.

3.1.2 Purpose. ML algorithms can be used for a wide variety of purposes, such as classifying an image or predicting the probability of an event. They are often used for the following tasks [85]:

- **Anomaly detection** is used to identify data samples that differ significantly from the majority of the data. These anomalies, which are also called outliers, are used in a wide range of applications including video surveillance, fraud detection in credit card transactions, or health monitoring with on-body sensors.
- **Classification** is the problem of categorizing unknown data points into categories seen during training. This is an inherently supervised process; the unsupervised equivalent of classification is clustering.
- **Clustering** groups data points that are similar according to a given metric. Small datasets can be clustered by manually labeling every instance, but for larger datasets that might be infeasible, which justifies the need for automatic labeling the instances (namely, clustering).
- **Dimensionality reduction** is the problem of reducing the number of variables in the input data. This can either be achieved by selecting only relevant variables (feature selection), or by creating new variables that represent multiple others (feature extraction).
- **Representation learning** attempts to find proper representations of input data for, e.g., feature detection, classification, clustering, encoding, or matrix factorization. This often also implies a dimensionality reduction.
- **Regression** is the problem of estimating how a so-called *dependent* variable changes in value when other variables change with a certain amount.

3.1.3 Method. Every effective ML algorithm needs a method that forces the algorithm to improve itself based on new input data so it can improve its accuracy. We identify five different groups of ML methods that distinguish themselves through the way the algorithm learns:

- **Evolutionary Algorithms (EAs) [57]** (and specifically **Genetic algorithms**) learn iteratively based on evolution. The model that actually solves the problem is represented by a set of properties, called its *genotype*. The performance of the model is measured using a score, calculated using a *fitness function*. After calculating the fitness score of all generated models, the next iteration creates new genotypes based on mutation and crossover of models that produce more accurate estimates. Genetic algorithms can be used to create other algorithms, such as neural networks, belief networks, decision trees, and rule sets.
- **Stochastic Gradient Descent (SGD)–based algorithms** minimize a loss function defined on the outputs of the model by adapting the model's parameters in the direction of the negative gradient (the multi-variable derivative of a function). The gradient descent is called stochastic, as the gradient is calculated from a randomly sampled subset of the training data. The loss function is typically a proxy for the actual error to be minimized; for example, the mean squared error between the model outputs and desired outputs in the case of a regression problem, or the negative log likelihood of the ground truth class according to the model in the case of classification. The typical training procedure then becomes:

  (1) Present a batch of randomly sampled training data.
  (2) Calculate the loss function of the model output and the desired output.
  (3) Calculate the gradient with respect to the model parameters.
  (4) Adjust the model parameters in the direction of the negative gradient, multiplied by a chosen learning rate.
  (5) Repeat

  SGD is the most commonly used training method for a variety of ML models.
  - **Support Vector Machines (SVMs)** map data points to high-dimensional vectors for classification and clustering purposes. For data points in a p-dimensional space, a (p-1)-dimensional hyperplane can be used as a classifier. A reasonable choice would be the hyperplane that properly separates the data points in two groups based on their labels by the largest possible margin. Sometimes special transformation equations (called kernels) are used to transform all data points to a different representation, in which it is easier to find such a hyperplane.
  - **Perceptrons [104]** are binary classifiers that label input vectors as "active" or "inactive." A perceptron assigns a weight to all inputs and then sums over the products of these weights and their input. The outcome of this is compared to a threshold to determine the label. Perceptron-based algorithms commonly use the entire batch of training data in their attempt to find a solution that is optimal for the whole set. They are binary, and therefore primarily used for binary classification.
  - **Artificial Neural Networks (ANNs)** are perceptron-based systems that consist of multiple layers: an input layer, one or more hidden layers, and an output layer. Each layer consists of nodes connected to the previous and next layers through edges with associated weights (usually called synapses). Unlike regular perceptrons, these nodes usually apply an activation function on the output to introduce non-linearities.

    The model is defined by the state of the entire network and can be changed by altering (1) the weights of the synapses, (2) the layout of the network, or (3) the activation function of nodes.

Because neural networks require a large number of nodes, the understandability of a neural network's *thought process* is lower compared to, e.g., decision trees.

Neural networks are extensively studied because of their ability to analyze enormous sets of data. They can be categorized into several subgroups based on network layout:

* **Deep Neural Networks (DNNs)**, are artificial neural networks that have many hidden layers. This allows the neural network to learn hierarchical feature abstractions of the data, with increasing abstraction the deeper you go in the network.
* **Convolutional Neural Networks (CNNs/ConvNets)** are deep, feed-forward neural networks that use convolution layers with nodes connected to only a few nodes in the previous layer. These values are then pooled using pooling layers. It can be seen as a way of recognizing abstract features in the data. The convolution makes the network consider only local data. This makes the represented algorithms spatially invariant, which is why they are sometimes called Space Invariant Artificial Neural Networks (SIANN). Chaining multiple of these convolution and pooling layers together can make the network capable of recognizing complicated constructs in big datasets. Examples of this are cats in images or the contextual meaning of a sentence in a paragraph.
* **Recurrent Neural Networks (RNNs)** keep track of a temporal state in addition to weights, which means that previous inputs of the network influence its current decisions. Recurrent synapses give the network a *memory*. This can help with discovering temporal patterns in data. Blocks of nodes in recurrent networks operate as cells with distinct memories and can store information for an arbitrarily long timespan.
* **Hopfield Networks** are a type of non-reflexive, symmetric recurrent neural network that have an *energy* related to every state of the network as a whole. They are guaranteed to converge on a local minimum after some number of network updates.
* **Self-Organizing Maps (SOMs)/Self-Organizing Feature Maps (SOFMs)** are neural networks that learn through unsupervised *competitive learning*, in which nodes compete for access to specific inputs. This causes the nodes to become highly specialized, which reduces redundancy. The iterations effectively move the map closer to the training data, which is the reason for its name. Some subtypes include the Time Adaptive Self-Organizing Map (TASOM, automatically adjust the learning rate and neighborhood size of each neuron independently), Binary Tree TASOM (BTASOM, tree of TASOM networks), and Growing Self-Organizing map (GSOM, identify a suitable map size in the SOM by starting with a minimal set of nodes and growing the map by heuristically adding new nodes at the periphery).
* **Stochastic Neural Networks** make use of stochastic transfer functions or stochastic weights, which allows them to escape the local minima that impede the convergence to a global minimum of normal neural networks. An example is a Boltzmann machine where each neuron output is represented as a binary value and the likelihood of the neuron firing depends on the network of other neurons.
* **Auto-encoders** are a type of neural network that are trained specifically to encode and decode data. Since auto-encoders are trained to perform decoding separately from encoding, the encoded version of the data is a form of dimensionality reduction of the data.
* **Generative Adversarial Networks (GAN)** are generative models that are trained using a minimax game between a generator and discriminator network [58]. The goal is

to train a neural network to generate data from a training set distribution. To achieve this, a discriminator neural network is trained at the same time to learn to discriminate between real dataset samples and generated samples by the generator. The discriminator is trained to minimize the classification errors, whereas the generator is trained to maximize the classification errors, in effect generating data that are indistinguishable from the real data.

- **Rule-Based Machine Learning (RBML) Algorithms [156]** use a set of rules that each represent a small part of the problem. These rules usually express a condition, as well as a value for when that condition is met. Because of the clear if-then relation, rules lend themselves to simple interpretation compared to more abstract types of ML algorithms, such as neural networks.
  - **Association Rule Learning** is a *rule-based machine learning* method that focuses on finding relations between different variables in datasets. Example relatedness metrics are *Support* (how often variables appear together), *Confidence* (how often a causal rule is true), and *Collective Strength* (inverse likelihood of the current data distribution if a given rule does not exist).
  - **Decision Trees**, sometimes called "CART" trees (after Classification And Regression Trees), use rule-based machine learning to create a set of rules and decision branches. Traversing the tree involves applying the rules at each step until a leaf of the tree is reached. This leaf represents the decision or classification for that input.
- **Topic Models (TM) [21]** are statistical models for finding and mapping semantic structures in large and unstructured collections of data, most often applied on text data.
  - **Latent Dirichlet Allocation** [22] constructs a mapping between documents and a probabilistic set of topics using the assumption that documents have few different topics and that those topics use few different words. It is used to learn what unstructured documents are about based on a few keywords.
  - **Latent Semantic Analysis (LSA)/Latent Semantic Indexing (LSI)** creates a big matrix of documents and topics in an attempt to classify documents or to find relations between topics. LSA/LSI assumes a Gaussian distribution for topics and documents. LSA/LSI does not have a way of dealing with words that have multiple meanings.
  - **Naive Bayes Classifiers** are relatively simple probabilistic classifiers that assume different features to be independent. They can be trained quickly using supervised learning but are less accurate than more complicated approaches.
  - **Probabilistic Latent Semantic Analysis (PLSA)/Probabilistic Latent Semantic Indexing (PLSI)** is the same as LSA/LSI, except that PLSA/PLSI assumes a Poisson distribution for topics and documents instead of the Gaussian distribution that is assumed by LSA/LSI. The reason is that a Poisson distribution appears to model the real world better [72]. Some subtypes include Multinomial Asymmetric Hierarchical Analysis (MASHA), Hierarchical Probabilistic Latent Semantic Analysis (HPLSA), and Latent Dirichlet Allocation (LDA).
- **Matrix Factorization** algorithms can be applied for identifying latent factors or find missing values in matrix-structured data. For example, many recommender systems are based on matrix factorization of the *User-Item Rating Matrix* to find new items users might be interested in, given their rating on other items [83]. Similarly factorizing a *Drug compound-Target Protein Matrix* is used for new drug discovery [63]. As this problem scales with $O(F^3)$ with F the dimensionality of the features, recent research focuses on scaling these methods to larger feature dimensions [142].

3.2 Hyperparameter Optimization

The performances of many of the algorithms presented in the previous sections are largely impacted by the choice of a multitude of algorithm hyperparameters. For example, in stochastic gradient descent, one has to choose the batch size, the learning rate, the initialization of the model, and so on. Often, the optimal values of these hyperparameters are different for each problem domain, ML model, and dataset.

There are several algorithms that can be used to automatically optimize the parameters of the machine learning algorithms and that can be re-used across different ML algorithm families.

These include:

- First-order algorithms that use at least one first-derivative of the function that maps the parameter value to the accuracy of the ML algorithm using that parameter. Examples are stochastic gradient descent (SGD) [24], stochastic dual coordinate ascent [136], or conjugate gradient methods [42, 69].
- Second-order techniques that use any second-derivative of the function that maps the parameter value to the accuracy of the ML algorithm using that parameter. Examples are Newton's method [120] (which requires computing the Hessian matrix, and is therefore generally infeasible), Quasi-Newton methods [28] (which approximate Newton's method by updating the Hessian by analyzing successive gradient vectors instead of recomputing the Hessian in every iteration), or L-BFGS [95].
- Coordinate descent [158] (also called *coordinate-wise minimization*), which minimizes at each iteration a single variable while keeping all other variables at their value of the current iteration.
- The Markov-Chain Monte-Carlo [26], which works by successively guessing new parameters randomly drawn from a normal multivariate solution centered on the old parameters and using these new parameters with a chance dependent on the likelihood of the old and the new parameters.
- A naive but often-used strategy is grid search, which exhaustively runs to a grid of potential values of each hyperparameter [88].
- Random search uses randomly chosen trials for sampling hyperparameter values, which often yields better results in terms of efficiency compared to grid search, finding better parameter values for the same compute budget [17].
- Bayesian hyperparameter optimization techniques use the Bayesian framework to iteratively sample hyperparameter values [146]. These model each trial as a sample from a Gaussian process (GP) and use the GP to choose the most informative samples in the next trial.

3.3 Combining Multiple Algorithms: Ensemble Methods

For some applications, a single model is not accurate enough to solve the problem. To alleviate this issue, multiple models can be combined in so-called *Ensemble Learning*. For example, when machine learning algorithms are performed on inherently distributed data sources and centralization is thus not an option, the setup requires training to happen in two separate stages: first in the local sites where the data are stored, and second in the global site that aggregates over the individual results of the first stage [77]. This aggregation can be achieved by applying ensemble methods in the global site.

Various different ways exist to perform ensembling, such as [50]:

- **Bagging** is the process of building multiple classifiers and combining them into one.

- **Boosting** is the process of training new models with the data that are misclassified by the previous models.
- **Bucketing** is the process of training many different models and eventually selecting the one that has the best performance.
- **Random Forests** [25] use multiple decision trees and averaging the prediction made by the individual trees to increase the overall accuracy. Different trees are given the same "voting power."
- **Stacking** is when multiple classifiers are trained on the dataset, and one new classifier uses the output of the other classifiers as input in an attempt to reduce the variance.
- **Learning Classifier Systems (LCSs)** is a modular system of learning approaches. An LCS iterates over data points from the dataset, completing the entire learning process in each iteration. The main idea is that an LCS has a limited number of rules. A genetic algorithm forces suboptimal rules out of the rule set. There are many different attributes that can drastically change the performance of an LCS depending on the dataset, including the Michigan-style vs. Pittsburgh-style architecture [113], supervised vs. reinforcement learning [81], incremental vs. batch learning [37], online vs. offline training, strength-based vs. accuracy-based [157], and complete mapping vs. best mapping.

3.4 Topologies

Another consideration for the design of a distributed machine learning deployment is the structure in which the computers within the cluster are organized. A deciding factor for the topology is the degree of distribution that the system is designed to implement. Figure 3 shows four possible topologies, in accordance with the general taxonomy of distributed communication networks by Baran [15].

Centralized systems (Figure 3(a)) employ a strictly hierarchical approach to aggregation, which happens in a single central location. *Decentralized systems* allow for intermediate aggregation, either with a replicated model that is consistently updated when the aggregate is broadcast to all nodes such as in tree topologies (Figure 3(b)) or with a partitioned model that is sharded over multiple parameter servers (Figure 3(c)). *Fully distributed systems* (Figure 3(d)) consist of a network of independent nodes that ensemble the solution together and where no specific roles are assigned to certain nodes.

There are several distinct topologies that have become popular choices for distributed machine learning clusters:

- **Trees.** Tree-like topologies have the advantage that they are easy to scale and manage, as each node only has to communicate with its parent and child nodes. For example, in the AllReduce [5] paradigm, nodes in a tree accumulate their local gradients with those from their children and pass this sum to their parent node to calculate a global gradient.
- **Rings.** In situations where the communication system does not provide efficient support for broadcast or where communication overhead needs to be kept to a minimum, ring topologies for AllReduce patterns simplify the structure by only requiring neighbor nodes to synchronize through messages. This is, e.g., commonly used between multiple GPUs on the same machine [76].
- **Parameter Server.** The Parameter Server paradigm (PS) [155] uses a decentralized set of workers with a centralized set of masters that maintain the shared state. All model parameters are stored in a shard on each parameter server, from which all clients read and write as a key-value store. An advantage is that all model parameters (within a shard) are in a global shared memory, which makes it easy to inspect the model. A disadvantage of the

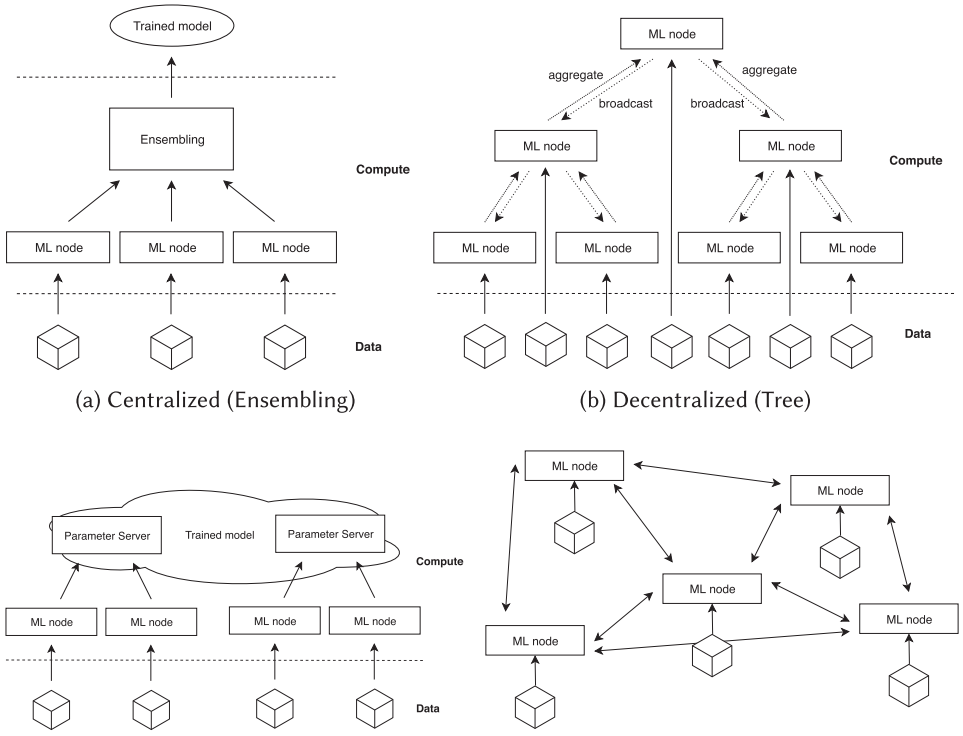

Fig. 3. Distributed machine learning topologies based on the degree of distribution.

topology is that the parameter servers can form a bottleneck, because they are handling all communication. To partially alleviate this issue, the techniques for bridging computation and communication mentioned in Section 3.5.2 are used.

- **Peer-to-Peer.** In contrast to centralized state, in the fully distributed model, every node has its own copy of the parameters and the workers communicate directly with each other. This has the advantage of typically higher scalability than a centralized model and the elimination of single points of failure in the system [52]. An example implementation of this model is a peer-to-peer network, in which nodes broadcast updates to all other nodes to form a data-parallel processing framework. Since full broadcast is typically prohibitive due to the volume of communication, Sufficient Factor Broadcasting (SFB) [94] has been proposed to reduce the communication overhead. The parameter matrix in SFB is decomposed into so-called sufficient factors, i.e., two vectors that are sufficient to reconstruct the update matrix. SFB only broadcasts these sufficient factors and lets the workers reconstruct the updates. Other models limit the degree of communication to less-frequent synchronization points while allowing the individual models to temporarily diverge. Gossip Learning [138] is built around the idea that models are mobile and perform independent random walks through the peer-to-peer network. Since this forms a data- and model-parallel processing framework, the models evolve differently and need to be combined through ensembling. In Gossip Learning, this happens continuously on the nodes by combining the current model with a limited cache of previous visitors.

3.5 Communication

As previously discussed, the need for more sophisticated machine learning-based setups quickly outgrows the capabilities of a single machine. There are several ways to partition the data and/or the program and to distribute these evenly across all machines. The choice of distribution, however, has direct implications on the amount of communication required to train the model.

3.5.1 Computation Time vs. Communication vs. Accuracy. When Distributed Machine Learning is used, one aims for the best accuracy at the lowest computation and communication cost. However, for complex ML problems, the accuracy usually increases with processing more training data, and sometimes by increasing the ML model size, hence increasing the computation cost. Parallelizing the learning can reduce computation time, as long as the communication costs are not becoming dominant. This can become a problem if the model being trained is not sufficiently large in comparison to the data. If the data are already distributed (e.g., cloud-native data), then there is no alternative to either moving the data or the computation.

Splitting up the dataset across different machines and training a separate model on a separate part of the dataset avoids communication, but this reduces the accuracy of the individual models trained on each machine. By ensembling all these models, the overall accuracy can be improved, However, the computation time is typically not much lower, since the individual models still have to take the same number of model update steps to converge.

By already synchronizing the different models during training (e.g., by combining the calculated gradients on all machines in case of gradient descent), the computation time can be reduced by converging faster to a local optimum. This, however, leads to an increase of communication cost as the model size increases.

Therefore, practical deployments require seeking the amount of communication needed to achieve the desired accuracy within an acceptable computation time.

3.5.2 Bridging Computation and Communication. To schedule and balance the workload, there are three concerns that have to be taken into account [161]:

- Identifying which tasks can be executed in parallel.
- Deciding the task execution order.
- Ensuring a balanced load distribution across the available machines.

After deciding on these three issues, the information between nodes should be communicated as efficiently as possible. There are several techniques that enable the interleaving of parallel computation and inter-worker communication. These techniques trade off fast/correct model convergence (at the top of the list found below) with faster/fresher updates (at the bottom of the list found below).

- **Bulk Synchronous Parallel (BSP)** is the simplest model in which programs ensure consistency by synchronizing between each computation and communication phase [161]. An example of a program following the BSP bridging model is MapReduce.

  An advantage is that serializable BSP ML programs are guaranteed to output a correct solution. A disadvantage is that finished workers must wait at every synchronization barrier until all other workers are finished, which results in overhead in the event of some workers progressing slower than others [34].
- **Stale Synchronous Parallel (SSP)** relaxes the synchronization overhead by allowing the faster workers to move ahead for a certain number of iterations. If this number is exceeded, then all workers are paused. Workers operate on cached versions of the data and only commit changes at the end of a task cycle, which can cause other workers to operate on stale

data. The main advantage of SSP is that it still enjoys strong model convergence guarantees. A disadvantage, however, is that when the staleness becomes too high (e.g., when a significant number of machines slows down), the convergence rates quickly deteriorate. The algorithm can be compared to Conits [166], used in distributed systems, because it specifies the data on which the workers are working and consistency is to be measured.

- **Approximate Synchronous Parallel (ASP)** limits how inaccurate a parameter can be. This contrasts with SSP, which limits how stale a parameter can be. An advantage is that, whenever an aggregated update is insignificant, the server can delay synchronization indefinitely. A disadvantage is that it can be hard to choose the parameter that defines which updates are significant and which are not [73].
- **Barrierless Asynchronous Parallel [65]/Total Asynchronous Parallel [73] (BAP/TAP)** lets worker machines communicate in parallel without waiting for each other. The advantage is that it usually obtains the highest possible speedup. A disadvantage is that the model can converge slowly or even develop incorrectly, because, unlike BSP and SSP, the error grows with the delay [65].

3.5.3 Communication Strategies. Communication is an important contributor to defining the performance and scalability of distributed processing [27]. Several communication management strategies [161] are used to spread and reduce the amount of data exchanged between machines:

- To prevent bursts of communication over the network (e.g., after a mapper is finished), continuous communication is used, such as in the state-of-the-art implementation Bösen [155].
- Neural networks are composed out of layers, the training of which (using the backpropagation gradient descent algorithm) is highly sequential. Because the top layers of neural networks contain the most parameters while accounting for only a small part of the total computation, Wait-free Backpropagation (WFBP) [171] was proposed. WFBP exploits the neural network structure by sending out the parameter updates of the top layers while still computing the updates for the lower layers, hence hiding most of the communication latency.
- Because WFBP does not reduce the communication overhead, hybrid communication (HybComm) [171] was proposed. Effectively, it combines Parameter Servers (PS) [155] with Sufficient Factor Broadcasting (SFB) [159], choosing the best communication method depending on the sparsity of the parameter tensor. See below for more information about PS (under Centralized Storage) and SFB (under Decentralized Storage).

3.6 Discussion

While machine learning and artificial intelligence is a discipline with a long history in computer science, recent advancements in technology have caused certain areas like neural networks to experience unprecedented popularity and impact on novel applications. As with many emerging topics, functionality has been the primary concern, and the non-functional aspects have only played a secondary role in the discussion of the technology. As a result, the community has only a preliminary understanding of how distributed ML algorithms and systems behave as a workload and which classes of problems have a higher affinity to a certain methodology when considering performance or efficiency.

However, as with similar topics like big data analytics, systems aspects are increasingly becoming more important as the technology matures and consumers become more mindful about resource consumption and return of investment. This has caused ML algorithms and systems to be increasingly more co-designed, i.e., adapting algorithms to make better use of systems resources and designing novel systems that support certain classes of algorithms better. We expect this trend

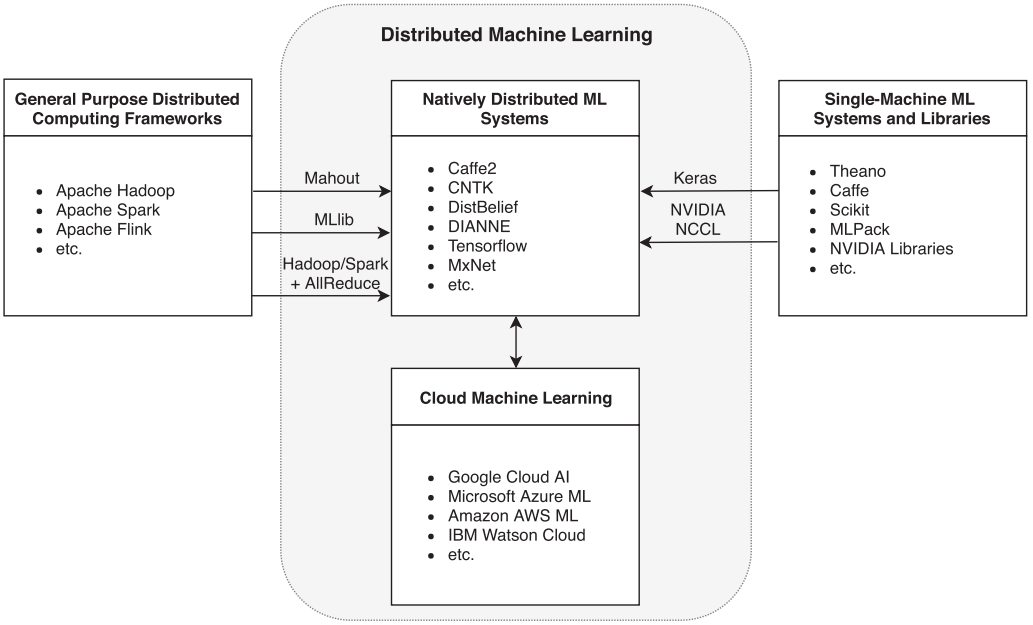

Fig. 4. Distributed machine learning ecosystem. Both general-purpose distributed frameworks and single-machine ML systems and libraries are converging towards distributed machine learning. Cloud emerges as a new delivery model for ML.

to continue and accelerate, eventually leading to a new wave of distributed machine learning systems that are more autonomous in their ability to optimize computation and distribution for given hardware resources. This would significantly lower the burden of adopting distributed machine learning in the same way that popular libraries have democratized machine learning in general by raising the level of abstraction from numerical computing to a simple and approachable templated programming style, or similar to the way that paradigms like MapReduce [44] have made processing of large datasets accessible.

4 THE DISTRIBUTED MACHINE LEARNING ECOSYSTEM

The problem of processing a large volume of data on a cluster of machines is not restricted to machine learning but has been studied for a long time in distributed systems and database research. As a result, some practical implementations use general-purpose distributed platforms as the foundation for distributed machine learning. Popular frameworks like Apache Spark [168, 169] have seized the opportunity of machine learning being an emerging workload and now provide optimized libraries (e.g., MLlib [98]). On the other end of the spectrum, purpose-built machine learning libraries that were originally designed to run on a single machine have started to receive support for execution in a distributed setting. For instance, the popular library Keras [35] received backends to run atop Google's Tensorflow [1] and Microsoft's CNTK [129]. Nvidia extended their machine learning stack with their Collective Communications Library (NCCL) [106], which was originally designed to support multiple GPUs on the same machine, but version 2 introduced the ability to run on multiple nodes [76]. The center this ecosystem (Figure 4) is inhabited by systems natively build for distributed machine learning and designed around a specific algorithmic and operational model, e.g., Distributed Ensemble Learning, Parallel Synchronous Stochastic Gradient Descent (SGD), or Parameter Servers. While the majority of these systems are intended to set up

and operated by the user and on-premise, there is an increasingly large diversity of machine learning services offered through a cloud delivery model, many centered around established distributed machine learning systems enhanced by a surrounding platform that makes the technology more consumable for data scientists and decision makers.

4.1 General Purpose Distributed Computing Frameworks

Distributed systems for processing massive amounts of data largely rely on utilizing a number of commodity servers, each of them with a relatively small storage capacity and computing power, rather than one expensive large server. This strategy has proven more affordable compared to using more expensive specialized hardware, as long as sufficient fault tolerance is built into the software, a concept that Google has pioneered [16] and that has increasingly found traction in the industry. Furthermore, the scale-out model offers a higher aggregate I/O bandwidth compared to using a smaller number of more powerful machines, since every node comes with its own I/O subsystem. This can be highly beneficial in data-intensive applications where data ingestion is a significant part of the workload [116].

4.1.1 Storage. The storage layer of existing frameworks is commonly based on the *Google File System (GFS)* [55] or comparable implementations. GFS is owned by and used within Google to handle all big data storage needs in the company. GFS splits up the data that are uploaded to the cluster into chunks, which are then distributed over the *chunk servers*. The chunks are replicated (the degree of replication is configurable and the default is three-way [55]) to protect the data from becoming unavailable in the event of machine failures. The data on the chunk servers can then be accessed by a user through contacting the master, which serves as a name node and provides the locations for every chunk of a file. The GFS architecture was adopted by an open-source framework called Hadoop [103], which was initially developed by Yahoo! and is now open source and maintained at the Apache Foundation. Its storage layer, named Hadoop File System or HDFS [141], started off as essentially a copy of the GFS design with only minor differences in nomenclature.

4.1.2 Compute. While the storage architecture has essentially converged to a block-based model, there exist many competing frameworks for scheduling and distributing tasks to compute resources with different features and trade-offs.

MapReduce is a framework (and underlying architecture) for processing data that was developed by Google [44] to process data in a distributed setting. The architecture consists of multiple phases and borrows concepts from functional programming. First, all data are split into tuples (called key-value pairs) during the *map phase*. This is comparable to a mapping of a second-order function to a set in functional programming. The map phase can be executed fully parallel, since there are no data dependencies between mapping a function to two different values in the set. Then, during the *shuffle phase*, these tuples are exchanged between nodes and passed on. This is strictly necessary, since aggregation generally has data dependencies and it has to be ensured that all tuples belonging to the same key are processed by the same node for correctness. In the subsequent *reduce phase*, the aggregation is performed on the tuples to generate a single output value per key. This is similar to a fold operation in functional programming, which rolls up a collection using a second-order function that produces a single result value. Fold, however, cannot be parallelized, since every fold step depends on the previous step. Shuffling the data and reducing by key is the enabler of parallelism in the reduce phase.

The main benefit of this framework is that the data can be distributed across a large number of machines while tasks of the same phase have no data dependencies and can therefore be executed entirely in parallel. Those same machines can be nodes in a GFS (or similar) storage cluster, so

instead of moving data to the program, the program can be moved to the data for an increase of data locality and better performance. The program is usually several orders of magnitude smaller to transfer over the wire, and is therefore much more efficient to pass around. Furthermore, in compliance with the idea of scale-out, MapReduce implements fault-tolerance in software by monitoring the health of the worker nodes through heartbeat messages and rescheduling tasks that failed to healthy nodes. Typically, the granularity of a task equals the size of a single block in the input dataset so a node failure should only affect a fraction of the overall application and the system is able to recover gracefully. Chu et al. [36] have mapped several ML algorithms to the MapReduce framework to exploit parallelism for multicore machines.

The MapReduce architecture is similar to the *Bulk-Synchronous Processing (BSP)* paradigm, which preceded it. However, there are some subtle differences. For instance, the MapReduce framework does not allow communication between worker nodes in the map phase. Instead, it only allows cross-communication during the shuffle phase, in between the map and reduce phases [115], for a reduction of synchronization barriers and an increase in parallelism. Goodrich et al. [59] have shown that all BSP programs can be converted into MapReduce programs. Pace [115], in turn, proposed that all MapReduce applications should be modeled as BSP tasks to combine the benefits of theoretical correctness of the BSP paradigm with the efficient execution of MapReduce.

MapReduce as a framework is proprietary to Google. The architecture behind it, however, has been recreated in the aforementioned open source Hadoop framework. It leverages HDFS where MapReduce uses GFS, but is similar in its overall architecture. Advanced variants have deliberated themselves from the strict tree topology of MapReduce data flows towards more flexible structures such as Forests (Dryad [75]) or generic Directed Acyclic Graphs (DAGs).

Apache Spark. MapReduce and Hadoop heavily rely on the distributed file system in every phase of the execution. Even intermediate results are stored on the storage layer, which can be a liability for iterative workloads that need to access the same data repeatedly. Transformations in linear algebra, as they occur in many ML algorithms, are typically highly iterative in nature. Furthermore, the paradigm of map and reduce operations is not ideal to support the data flow of iterative tasks, since it essentially restricts it to a tree-structure [86]. Apache Spark has been developed in response to this challenge. It is capable of executing a directed acyclic graph of transformations (like mappings) and actions (like reductions) fully in memory [137]. Because of its structure, Spark can be significantly faster than MapReduce for more complex workloads. When, for example, two consecutive map phases are needed, two MapReduce tasks would need to be executed, both of which would need to write all (intermediate) data to disk. Spark, however, can keep all the data in memory, which saves expensive reads from the disk.

The data structure that Spark was originally designed around is called a *Resilient Distributed Dataset (RDD)*. Such datasets are read-only, and new instances can only be created from data stored on the disk or by transforming existing RDDs [167]. The *Resilient* part comes into play when the data are lost: Each RDD is given a lineage graph that shows what transformations have been executed on it. This lineage graph ensures that, if some data are lost, Spark can trace the path the RDD has followed from the lineage graph and recalculate any lost data. It is important that the lineage graph does not contain cycles (i.e., is a Directed Acylic Graph). Otherwise, Spark would run into infinite loops and be unable to recreate the RDD. In practice, the need for re-computation as a result of data loss due to node failure can lead to ripple effects [167]. Spark allows for checkpointing of data to prevent extensive re-computation. Checkpoints have to be explicitly requested and essentially materialize the intermediate state while truncating the RDD lineage graph. Systems like TR-Spark [163] have automated the generation of checkpoints to make Spark able to run on transient resources where interruption of the execution has to be considered the norm.

Apache Spark also includes MLlib, a scalable machine learning library that implements many ML algorithms for classification, regression, decision trees, clustering, and topic modeling. It also provides several utilities for building ML workflows, implementing often-used feature transformations, hyperparameter tuning, and so on. As MLlib uses Spark's APIs, it immediately benefits from the scale-out and failure resilience features of Spark. MLLib relies on the Scala linear algebra package Breeze [64], which in turn utilizes netlib-java [98] for optimization, a bridge for libraries such as BLAS [20] and LAPACK [9], which are widely used in high-performance computing.

4.2 Natively Distributed Machine Learning Systems

As a result of the rising popularity of machine learning in many applications, several domain-specific frameworks have been developed around specific distribution models. In this section, the characteristics of the most popular implementations are summarized.

4.2.1 Distributed Ensemble Learning. Many generic frameworks and ML libraries have limited support for distributed training, even though they are fast and effective on a single machine. One way to achieve distribution with these frameworks is through training separate models for subsets of the available data. At prediction time, the outputs of those instances can then be combined through standard ensemble model aggregation [111].

Models that follow this strategy are not dependent on any specific library. They can be orchestrated using existing distribution frameworks (such as MapReduce [44]). The training process involves training individual models on independent machines in parallel. Neither orchestration nor communication are necessary once training has started. Training on m machines with m subsets of the data results in m different models. Each of these can use separate parameters or even algorithms. At prediction time, all trained models can then be run on new data, after which the output of each one is aggregated. This can once again be distributed if needed.

One large drawback is that this method is dependent on proper subdivision of the training data. If large biases are present in the training sets of some of the models, then those instances could cause biased output of the ensemble. If the data are divided manually, then it is paramount to ensure independence and identical distribution of the data (i.i.d.). If, however, the dataset is inherently distributed, then this is not straightforward to achieve.

There is a large number of existing frameworks available for this method, as any machine learning framework can be used. Some popular implementations use Tensorflow [1], MXNet [33], and PyTorch [117].

4.2.2 Parallel Synchronous Stochastic Gradient Descent. Synchronized parallelism is often the most straightforward to program and reason about. Existing distribution libraries (such as Message Passing Interface (MPI) [62]) can typically be reused for this purpose. Most approaches rely on the AllReduce operation [5] where the compute nodes are arranged in a tree-like topology. Initially, each node calculates a local gradient value, accumulates these with the values received from its children and sends these up to its parent (reduce phase). Eventually, the root node obtains the global sum and broadcasts this back down to the leaf nodes (broadcast phase). Then each node updates its local model with regard to the received global gradient.

Baidu AllReduce uses common high performance computing technology (mainly MPI and its AllReduce operation) to iteratively train SGD models on separate mini-batches of the training data [56]. AllReduce is used to apply each of the workers' gradients onto the last common model state after each operation and then propagate the result of that operation back to each worker. This is an inherently synchronous process, blocking on the result of each worker's training iteration before continuing to the next.

Baidu includes a further optimization from Patarasuk and Yuan [118] in this process, called a Ring AllReduce, to reduce the required amount of communication. By structuring the cluster of machines as a ring (with each node having only two neighbors) and cascading the reduction operation, it is possible to utilize all bandwidth optimally. The bottleneck, then, is the highest latency between neighboring nodes.

Baidu claims linear speedup when applying this technique to train deep learning networks. However, it has only been demonstrated on relatively small clusters (five nodes each, though each node has multiple GPUs that communicate with each other through the same system). The approach lacks fault tolerance by default, as no node in the ring can be missed. This could be counteracted using redundancy (at cost of efficiency). If this is not done, however, then the scalability of the method is bounded by the probability of all nodes being available. This probability can be low when using large numbers of commodity machines and networking, which is needed to facilitate big data. Baidu's system has been integrated into Tensorflow as an alternative to the built-in Parameter Server–based approach (described below).

Horovod [131] takes a very similar approach to that of Baidu: It adds a layer of AllReduce-based MPI training to Tensorflow. One difference is that Horovod uses the NVIDIA Collective Communications Library (NCCL) for increased efficiency when training on (Nvidia) GPUs. This also enables use of multiple GPUs on a single node. Data-parallelizing an existing Tensorflow model is relatively simple, since only a few lines of code need to be added, wrapping the default Tensorflow training routine in a distributed AllReduce operation. When benchmarked on Inception v4 [148] and ResNet-101 [68] using 128 GPUs, the average GPU utilization is about 88%, compared to about 50% in Tensorflow's Parameter Server approach. However, Horovod lacks fault tolerance (just like in Baidu's approach) and therefore suffers from the same scalability issues [53].

Caffe2 (primarily maintained by Facebook) distributes ML through, once again, AllReduce algorithms. It does this by using NCCL between GPUs on a single host and custom code between hosts based on Facebook's Gloo [47] library to abstract away different interconnects. Facebook uses Ring AllReduce (which offers better bandwidth and parallelism guarantees) but also recursive halving and doubling (a divide-and-conquer approach that offers better latency guarantees). According to their paper, this improves performance in latency-limited situations, such as for small buffer sizes and large server counts. He et al. [68] managed to train ResNet-50 in the span of one hour [61] using this approach, achieving linear scaling with the number of GPUs. They achieved 90% efficiency, measured up to 352 GPUs. However, once again, no fault-tolerance is present.

CNTK or The Microsoft Cognitive Toolkit offers multiple modes of data-parallel distribution. Many of them use the Ring AllReduce tactic as previously described, making the same trade-off of linear scalability over fault-tolerance. The library offers two innovations:

- **1-bit stochastic gradient descent** (Seide et al. [130]) is an implementation of SGD that quantizes training gradients to a single bit per value. This reduces the number of bits that need to be communicated when doing distributed training by a large constant factor.
- **Block-momentum SGD** (Chen and Huo [31]) divides the training set into m blocks and n splits. Each of the n machines trains a split on each block. Then the gradients calculated for all splits within a block are averaged to obtain the weights for the block. Finally, the block updates are merged into the global model while applying block-level momentum and learning rate.

When benchmarked on a Microsoft speech LSTM, average speedups of 85%+ are achieved for small numbers of GPUs (up to 16), but scalability drops significantly (below 70%) when scaling

past that. However, the direct comparison of this number to the other synchronous frameworks' results is questionable, as the dependency structure of an LSTM is significantly different than that of an ordinary DNN due to the introduction of temporal state [139].

4.2.3 Parallel Asynchronous Stochastic Gradient Descent and Parameter Servers. Asynchronous approaches tend be more complex to implement, and it can be more difficult to trace and debug runtime behavior. However, asynchronism alleviates many problems that occur in clusters with high failure rates or inconsistent performance due to the lack of frequent synchronization barriers.

DistBelief [43] is one of the early practical implementations of large-scale distributed ML, and it was developed by Google. They encountered the limitations of GPU training and built DistBelief to counteract them. DistBelief supports data- and model-parallel training on tens of thousands of CPU cores (though GPU support was later introduced as well [2]). They reported a speedup of more than 12× when using 81 machines training a huge model with 1.7B parameters.

To achieve efficient model-parallelism, DistBelief exploits the structure of neural networks and defines a model as a computation graph where each node implements an operation transforming inputs to outputs. Every machine executes the training of a part of the computation graph's nodes, which can span subsets of multiple layers of the neural network. Communication is only required at those points where a node's output is used as the input of a node trained by another machine. Partitioning the model across a cluster is transparent and requires no structural modifications. However, the efficiency of a given partitioning is greatly affected by the architecture of the model and requires careful design. For example, locally connected models lend themselves better for model-parallelism because of limited cross-partition communication. In contrast, fully connected models have more substantial cross-partition dependencies and are therefore harder to efficiently distribute through DistBelief.

To further parallelize model training, data parallelism is applied on top of the model parallelism. A centralized sharded Parameter Server is used to allow each of a set of model replicas (which may be model-parallel internally) to share parameters. DistBelief supports two different methods of data parallelism, both of which are resilient to processing speed variance between model replicas as well as replica failure:

- **Downpour Stochastic Gradient Descent** is an asynchronous alternative to the inherently sequential SGD. Each replica of the model fetches the latest model parameters from the Parameter Server every n_{fetch} steps, updates these parameters in accordance with the model, and pushes the tracked parameter gradients to the Parameter Server every n_{push} steps. The parameters n_{fetch} and n_{push} can be increased to achieve lower communication overhead. Fetching and pushing can happen as a background process, allowing training to continue.

  Downpour SGD is more resilient to machine failures than SGD, as it allows the training to continue even if some model replicas are off-line. However, the optimization process itself becomes less predictable due to parameters that are out of sync. The authors found relaxing consistency requirements to be remarkably effective, but offer no theoretical support for this. Tactics that contribute to robustness are the application of adaptive learning rates through AdaGrad [45] and *warm starting* the model through training a single model replica for a while before scaling up to the full number of machines. The authors make note of the absence of stability issues after applying these.
- **Distributed L-BGFS** makes use of an external coordinator process that divides training work between model replicas, as well as some operations on the parameters between the parameter server shards. Training happens through L-BGFS, as is clear from the name.

Each of the shards of the Parameter Server hold a fraction of the parameter space of a model. The model replicas pull the parameters from all shards and each parallelized part of the model only retrieves those parameters that it needs.

Performance improvements are high, but the methodology is very expensive in terms of computational complexity. While the best speedup (downpour SGD with AdaGrad) achieved an 80% decrease in training time on ImageNet; this was achieved by using more than 500 machines and more than 1K CPU cores. It has to be noted that DistBelief did not support distributed GPU training at the time of Dean et al. [43], which could reduce the required resources significantly and is used in fact by almost all other implementations mentioned in this section.

DIANNE (DIstributed Artificial Neural NEtworks) [39] is a Java-based distributed deep learning framework using the Torch native backend for executing the necessary computations. It uses a modular OSGi-based distribution framework [154] that allows to execute different components of the deep learning system on different nodes of the infrastructure. Each basic building block of a neural network can be deployed on a specific node, hence enabling model-parallelism. DIANNE also provides basic learner, evaluator, and parameter server components that can be scaled and provide a downpour SGD implementation similar to DistBelief.

Tensorflow [1, 2] is the evolution of DistBelief, developed to replace DistBelief within Google. It borrows the concepts of a computation graph and parameter server from it. It also applies subsequent optimizations to the parameter server model, such as optimizations for training convolutional neural networks [34] and innovations regarding consistency models and fault tolerance [92, 93]. Unlike DistBelief, TensorFlow was made available as open source software.

TensorFlow represents both model algorithms and state as a dataflow graph, of which the execution can be distributed. This facilitates different parallelization schemes that can take, e.g., state locality into account. The level of abstraction of the dataflow graph is mathematical operations on tensors (i.e., *n*-dimensional matrices). This in contrast to DistBelief, which abstracts at the level of individual layers. Consequently, defining a new type of neural network layer in Tensorflow requires no custom code—it can be represented as a subgraph of a larger model, composed of fundamental math operations. A Tensorflow model is first defined as a symbolic dataflow graph. Once this graph has been constructed, it is optimized and then executed on the available hardware. This execution model allows Tensorflow to tailor its operations towards the types of devices available to it. When working with, e.g., GPUs or TPUs (Tensor Processing Units [80]), Tensorflow can take into account the asynchronicity and intolerance or sensitivity to branching that is inherent to these devices, without requiring any changes to the model itself.

Shi and Chu [138] show Tensorflow achieving about 50% efficiency on four-node, InfiniBand-connected cluster training of ResNet-50 [68] and about 75% efficiency on GoogleNet [147], showing that the communication overhead plays an important role and also depends on architecture of the neural network to optimize.

MXNet [33] uses a strategy very similar to that of Tensorflow: Models are represented as dataflow graphs, which are executed on hardware that is abstracted away and coordinated by using a parameter server. However, MXNet also supports the imperative definition of dataflow graphs as operations on n-dimensional arrays, which simplifies the implementation of certain kinds of networks.

MXNet's Parameter Server, KVStore, is implemented on top of a traditional key-value store. The KVStore supports pushing key-value pairs from a device to the store, as well as pulling the current value of a key from the store. There is support for user-defined update logic that is executed when a new value is pushed. The KVStore can also enforce different consistency models (currently limited

to sequential and eventually consistent execution). It is a two-tier system: Updates by multiple threads and GPUs are merged on the local machine before they are pushed to the full cluster. The KVStore abstraction theoretically enables the implementation of (stale-)synchronicity, although only an asynchronous implementation is present at the time of writing.

On a small cluster of 10 machines equipped with a GPU, MXNet achieves almost linear speedup compared to a single machine when training GoogleNet [147] with more than 10 passes over the data [33].

DMTK or the Distributed Machine Learning Toolkit [102] from Microsoft includes a Parameter Server called *Multiverso*. This can be used together with CNTK to enable Asynchronous SGD instead of the default Allreduce-based distribution in CNTK.

4.2.4 Parallel Stale-synchronous Stochastic Gradient Descent.

Petuum [160] aims to provide a generic platform for any type of machine learning (as long as it is iteratively convergent) on big data and big models (hundreds of billions of parameters). It supports data- and model-parallelism. The Petuum approach exploits ML's error tolerance, dynamic structural dependencies, and non-uniform convergence to achieve good scalability on large datasets and models. This is in contrast to, for example, Spark, which focuses on fault tolerance and recovery. The platform uses stale synchronicity to exploit inherent tolerance of machine learning against errors, since a minor amount of staleness will only have minor effects on convergence. Dynamic scheduling policies are employed to exploit dynamic structural dependencies, which helps minimize parallelization error and synchronization cost. Finally, unconverged parameter prioritization takes advantage of non-uniform convergence by reducing computational cost on parameters that are already near optimal.

Petuum uses the Parameter Server paradigm to keep track of the parameters of the model being trained. The Parameter Server is also responsible for maintaining the staleness guarantees. In addition, it exposes a scheduler that lets the model developer control the ordering of parallelized model updates.

When developing a model using Petuum, developers have to implement a method named *push*, which is responsible for each of the parallelized model training operations. Its implementation should pull the model state from the parameter server, run a training iteration, and push a gradient to the parameter server. Petuum by default manages the scheduling aspect and the parameter merging logic automatically, so data-parallel models do not require any additional operations. However, if model-parallelism is desired, the schedule method (which tells each of the parallel workers what parameters they need to train) and the pull method (which defines the aggregation logic for each of the generated parameter gradients) need to be implemented as well.

Petuum provides an abstraction layer that also allows it to run on systems using YARN (the Hadoop job scheduler) and HDFS (the Hadoop file system), which simplifies compatibility with pre-existing clusters.

4.2.5 Parallel Hybrid-synchronous SGD. Both synchronous and asynchronous approaches have some significant drawbacks, as is explored by Chen et al. [30]. A few frameworks attempt to find a middle ground instead that combines some of the best properties of each model of parallelism and diminishes some of the drawbacks.

MXNet-MPI [96] takes an approach to distributed ML (using a modified version of MXNet as a proof of concept) that combines some of the best aspects of both asynchronous (Parameter Server) and synchronous (MPI) implementations. The idea here is to use the same architecture as described in the MXNet section. Instead of having single workers communicate with the parameter server, however, those workers are clustered together into groups that internally apply synchronous SGD

over MPI with AllReduce. This has the benefit of easy linear scalability of the synchronous MPI approach and fault tolerance of the asynchronous Parameter Server approach.

4.3 Machine Learning in the Cloud

Several cloud operators have added machine learning as a service to their cloud offerings. Most providers offer multiple options of executing machine learning tasks in their clouds, ranging from IaaS-level services (VM instances with pre-packaged ML software) to SaaS-level solutions (Machine Learning as a Service). Much of the technology offered are standard distributed machine learning systems and libraries. Among other things, Google's Cloud Machine Learning Engine offers support for TensorFlow and even provides TPU instances [60]. Microsoft Azure Machine Learning allows model deployment through Azure Kubernetes, through a batch service, or by using CNTK VMs [101]. As a competitor to Google's TPUs, Azure supports accelerating ML applications through FPGAs [114]. Amazon AWS has introduced SageMaker, a hosted service for building and training machine learning models in the cloud. The service includes support for TensorFlow, MXNet, and Spark [7]. IBM has bundled their cloud machine learning offerings under the Watson brand [74]. Services include Jupyter notebooks, Tensorflow, and Keras. The cloud-based delivery model is becoming more important, as it reduces the burden of entry into designing smart applications that facilitate machine learning techniques. However, the cloud is not only a consumer of distributed machine learning technology but is also fueling the development of new systems and approaches back to the ecosystem to handle the large scale of the deployments.

5 CONCLUSIONS AND CURRENT CHALLENGES

Distributed Machine Learning is a thriving ecosystem with a variety of solutions that differ in architecture, algorithms, performance, and efficiency. Some fundamental challenges had to be overcome to make distributed machine learning viable in the first place, such as finding mechanisms to efficiently parallelize the processing of data while combining the outcome into a single coherent model. Now that there are industry-grade systems available and in view of the ever-growing appetite for tackling more complex problems with machine learning, distributed machine learning is increasingly becoming the norm and single-machine solutions the exception, similar to how data processing in general had developed in the past decade. There are, however, still many open challenges that are crucial to the long-term success of distributed machine learning.

5.1 Performance

A trade-off that is seen frequently is the reduction of wall-clock time at the expense of total aggregate processing time (i.e., decreased efficiency) by adding additional resources. When compute resources are affordable enough, many real-world use cases of machine learning benefit most from being trained rapidly. The fact that this often implies a large increase in total compute resources and the associated energy consumption is not considered important as long as a model saves more money than it costs to train. A good example of this is found in Dean et al. [43], where wall-clock time speedup factors are achieved by increasing the number of machines quadratically or worse. It still delivered Google competitive advantage for years. Distributed use of GPUs, as in Tensorflow, has better properties, but often still exhibits efficiency below 75%. These performance concerns are much less severe in the context of synchronous SGD-based frameworks, which often do achieve linear speedups in benchmarks. However, most of these benchmarks test at most a few hundred machines, whereas the scale at which, e.g., DistBelief, is demonstrated, can be two orders of magnitude larger. The research community could clearly benefit from more independent studies that report on the performance and scalability of these systems for larger

and more realistic applications, and that could provide valuable insights to guide research into workload optimization and system architecture.

5.2 Fault Tolerance

Synchronous AllReduce-based approaches seem to scale significantly better than the parameter server approach (up to a certain cluster size), but suffer from a lack of fault-tolerance: Failure of a single machine blocks the entire training process. At smaller scales, this might still be a manageable problem. However, past a certain number of nodes, the probability of any node being unavailable becomes high enough to result in near-continuous stalling. Common implementations of these HPC-inspired patterns, such as MPI and NCCL, lack fault-tolerance completely. Although there are efforts to counteract some of this, production-ready solutions are lacking. Some of the described implementations allow for checkpointing to counteract this, but significant effort is necessary to enable true fault-tolerance, as is described in Amatya et al. [6]. It is also possible to reduce the probability of failure for each individual node, but this requires very specific hardware that is expensive and not generally available in commodity scale-out data centers or in the cloud. Asynchronous implementations do not suffer from this problem as much. They are designed to explicitly tolerate straggling [41] (slow-running) and failing nodes, with only minimal impact on training performance. The question for ML operators, then, is whether they prefer performance or fault tolerance, and whether they are constrained by either one. Hybrid approaches even offer a way to customize these characteristics, although they are not frequently found in use yet. It would be interesting to see whether an even better approach exists, or whether there is an efficient way to implement fault-tolerant AllReduce.

5.3 Privacy

There are scenarios in which it is beneficial or even mandatory to isolate different subsets of the training data from each other [79]. The furthest extent of this is when a model needs to be trained on datasets that each live on different machines or clusters and may under no circumstance be co-located or even moved. Peer-to-peer topologies like Gossip Learning [112] fully embrace this principle.

Another approach to training models in a privacy-sensitive context is the use of a distributed ensemble model. This allows perfect separation of the training data subsets, with the drawback that a method needs to be found that properly balances each trained model's output for an unbiased result.

Parameter server–based systems can be useful in the context of privacy, as the training of a model can be separated from the training result. Abadi et al. [3] discuss several algorithms that are able to train models efficiently while maintaining differential privacy. These parameter server–based systems assume that no sensitive properties of the underlying data leak into the model itself, which turns out to be difficult in practice. Recently, Bagdasaryan et al. [12] showed that it is possible for attackers to implement a back door into the joint model.

Federated learning systems can be deployed where multiple parties jointly learn an accurate deep neural network while keeping the data itself local and confidential. Privacy of the respective data was believed to be preserved by applying differential privacy, as shown by Shokri and Shmatikov [140] and McMahan et al. [97]. However, Hitaj et al. [71] devised an attack based on GANs, showing that record-level differential privacy is generally ineffective in federated learning systems.

Additionally, it is possible to introduce statistical noise into each subset of the training data with the intention of rendering its sensitive characteristics unidentifiable to other parties. Balcan et al. [14] touch on this subject but make it clear that the resulting privacy in this scenario is dependent

on the amount of statistical queries required to learn the dataset. This puts an upper bound on usefulness of the model itself.

For a more in-depth discussion on privacy in distributed deep learning, we refer to Vepakomma et al. [153]. In conclusion, while theoretical results exist, current frameworks do not offer much support for even basic forms of privacy. It could be interesting to investigate fundamental approaches to facilitate distributed privacy, which could then be integrated into the currently popular frameworks.

5.4 Portability

With the proliferation of machine learning, in particular deep learning, a myriad of different libraries and frameworks for creating and training neural networks is established. However, once trained, one is often stuck to the framework at hand to deploy the model in production, as they all use a custom format to store the results. For example, Tensorflow [2] uses a SavedModel directory, which includes a protocol buffer defining the whole computation graph. Caffe [78] also uses a binary protocol buffer for storing saved models, but with a custom schema. Theano [18] uses pickle to serialize models represented by Python objects, and PyTorch [117] has a built-in save method that serializes to a custom ASCII or binary format.

Portability also becomes increasingly important with respect to the hardware platform on which one wants to deploy. Although the x86_64 and ARM processor architectures are mainstream to execute applications in the server and mobile devices market, respectively, we witness a shift towards using GPU hardware for efficiently executing neural network models [108]. As machine learning models become more widespread, we also see more development of custom ASICs such as TPUs [128] in Google Cloud or dedicated neural network hardware in the latest iPhone [11]. This diversification makes it more difficult to make sure that your trained model can run on any of these hardware platforms.

A first step towards portability is the rise of a couple of framework-independent specifications to define machine learning models and computation graphs. The Open Neural Network Exchange (ONNX) format defines a protocol buffer schema that defines an extensible computation graph model as well as definitions for standard operators and data types. Currently, ONNX is supported out of the box by frameworks such as Caffe, PyTorch, CNTK, and MXNet, and converters exist, e.g., for TensorFlow. Similar efforts for a common model format specification are driven by Apple with their Core ML format [10] and the Khronos Group with the Neural Network Exchange Format [150].

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Retrieved from https://www.tensorflow.org/.

[2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 265–283.

[3] Martin Abadi, Andy Chu, Ian Goodfellow, Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep learning with differential privacy. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (ACM CCS'16)*. 308–318. Retrieved from https://arxiv.org/abs/1607.00133.

[4] Adapteva, Inc. 2017. E64G401 Epiphany 64-core Microprocessor Datasheet. Retrieved from http://www.adapteva.com/docs/e64g401_datasheet.pdf.
[5] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. 2014. A reliable effective terascale linear learning system. *J. Mach. Learn. Res.* 15, 1 (2014), 1111–1133.
[6] Vinay Amatya, Abhinav Vishnu, Charles Siegel, and Jeff Daily. 2017. What does fault tolerant deep learning need from MPI? *CoRR* abs/1709.03316 (2017). arxiv:1709.03316 http://arxiv.org/abs/1709.03316.
[7] Amazon Web Services. 2018. Amazon SageMaker. Retrieved from https://aws.amazon.com/sagemaker/developer-resources/.
[8] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. 2016. Deep speech 2 : End-to-end speech recognition in English and Mandarin. In *Proceedings of the 33rd International Conference on Machine Learning*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. PMLR, New York, New York, 173–182. Retrieved from http://proceedings.mlr.press/v48/amodei16.html.
[9] Edward Anderson, Zhaojun Bai, Jack Dongarra, Anne Greenbaum, Alan McKenney, Jeremy Du Croz, Sven Hammarling, James Demmel, C. Bischof, and Danny Sorensen. 1990. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 2–11.
[10] Apple. 2017. Core ML Model Format Specification. Retrieved from https://apple.github.io/coremltools/coremlspecification/.
[11] Apple. 2018. A12 Bionic. Retrieved from https://www.apple.com/iphone-xs/a12-bionic/.
[12] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. 2018. How to backdoor federated learning. *arXiv preprint arXiv:1807.00459* (2018).
[13] Drew Bagnell and Andrew Y. Ng. 2006. On local rewards and scaling distributed reinforcement learning. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 91–98.
[14] Maria-Florina Balcan, Avrim Blum, Shai Fine, and Yishay Mansour. 2012. Distributed learning, communication complexity and privacy. In *Proceedings of the Conference on Learning Theory*. 26–1.
[15] Paul Baran. 1962. *On Distributed Communication Networks*. Rand Corporation.
[16] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. 2003. Web search for a planet: The Google cluster architecture. *IEEE Micro* 2 (2003), 22–28.
[17] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* 13 (Feb. 2012), 281–305.
[18] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math compiler in Python. In *Proceedings of the 9th Python in Science Conference*, Vol. 1.
[19] Philip A. Bernstein and Eric Newcomer. 2009. *Principles of Transaction Processing*. Morgan Kaufmann.
[20] L. Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R. Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2 (2002), 135–151.
[21] David M. Blei. 2012. Probabilistic topic models. *Commun. ACM* 55, 4 (2012), 77–84.
[22] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet allocation. *J. Mach. Learn. Res.* 3, Jan. (2003), 993–1022.
[23] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to end learning for self-driving cars. *CoRR* abs/1604.07316 (2016). arxiv:1604.07316 http://arxiv.org/abs/1604.07316.
[24] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the COMPSTAT'2010*. Springer, 177–186.
[25] Leo Breiman. 2001. Random forests. *Mach. Learn.* 45, 1 (1 Oct. 2001), 5–32.
[26] Stephen Brooks. 1998. Markov chain Monte Carlo method and its application. *J. Roy. Statist. Soc.: Series D (the Statist.)* 47, 1 (1998), 69–100.
[27] Rajkumar Buyya et al. 1999. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall, Upper SaddleRiver, NJ, 999.

[28] Richard H. Byrd, Samantha L. Hansen, Jorge Nocedal, and Yoram Singer. 2016. A stochastic quasi-Newton method for large-scale optimization. *SIAM J. Optim.* 26, 2 (2016), 1008–1031.
[29] K. Canini, T. Chandra, E. Ie, J. McFadden, K. Goldman, M. Gunter, J. Harmsen, K. LeFevre, D. Lepikhin, T. L. Llinares, et al. 2012. Sibyl: A system for large scale supervised machine learning. *Tech. Talk* 1 (2012), 113.
[30] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Józefowicz. 2016. Revisiting distributed synchronous SGD. *CoRR* abs/1604.00981 (2016). arxiv:1604.00981 http://arxiv.org/abs/1604.00981.
[31] Kai Chen and Qiang Huo. 2016. Scalable training of deep learning machines by incremental block training with intra-block parallel optimization and blockwise model-update filtering. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'16)*. IEEE, 5880–5884.
[32] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGPLAN Not.* 49, 4 (2014), 269–284.
[33] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR* abs/1512.01274 (2015). arxiv:1512.01274 http://arxiv.org/abs/1512.01274.
[34] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, 571–582. Retrieved from https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi.
[35] François Chollet et al. 2015. Keras. Retrieved from https://keras.io/.
[36] Cheng-Tao Chu, Sang K. Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y. Ng. 2007. Map-reduce for machine learning on multicore. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 281–288.
[37] Scott H. Clearwater, Tze-Pin Cheng, Haym Hirsh, and Bruce G. Buchanan. 1989. Incremental batch learning. In *Proceedings of the 6th International Workshop on Machine Learning*. Elsevier, 366–370.
[38] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. 2013. Deep learning with COTS HPC systems. In *Proceedings of the International Conference on Machine Learning*. 1337–1345.
[39] Elias De Coninck, Steven Bohez, Sam Leroux, Tim Verbelen, Bert Vankeirsbilck, Pieter Simoens, and Bart Dhoedt. 2018. DIANNE: A modular framework for designing, training and deploying deep neural networks on heterogeneous distributed infrastructure. *J. Syst. Softw.* 141 (2018), 52–65.
[40] George F. Coulouris, Jean Dollimore, and Tim Kindberg. 2005. *Distributed Systems: Concepts and Design*. Pearson Education.
[41] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, et al. 2014. Exploiting bounded staleness to speed Up big data analytics. In *Proceedings of the USENIX Annual Technical Conference*. 37–48.
[42] Yu-Hong Dai and Yaxiang Yuan. 1999. A nonlinear conjugate gradient method with a strong global convergence property. *SIAM J. Optim.* 10, 1 (1999), 177–182.
[43] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems, Vol. 1 (NIPS'12)*. Curran Associates Inc., 1223–1231. Retrieved from http://dl.acm.org/citation.cfm?id=2999134.2999271.
[44] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Operating Systems Design & Implementation, Vol. 6 (OSDI'04)*. USENIX Association, 10–10.
[45] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.* 12 (July 2011), 2121–2159. Retrieved from http://dl.acm.org/citation.cfm?id=1953048.2021068.
[46] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* 34, 3 (2002), 375–408.
[47] Facebook. 2017. Gloo. Retrieved from https://github.com/facebookincubator/gloo.
[48] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. 2011. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW'11)*. IEEE, 109–116.
[49] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Not.*, Vol. 47. ACM, 37–48.
[50] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. 2014. Do we need hundreds of classifiers to solve real world classification problems. *J. Mach. Learn. Res* 15, 1 (2014), 3133–3181.

[51] Michael J. Flynn. 1972. Some computer organizations and their effectiveness. *IEEE Trans. Comput.* 100, 9 (1972), 948–960.
[52] Ian Foster and Adriana Iamnitchi. 2003. On death, taxes, and the convergence of peer-to-peer and grid computing. In *Proceedings of the International Workshop on Peer-to-Peer Systems*. Springer, 118–128.
[53] Ermias Gebremeskel. 2018. Analysis and comparison of distributed training techniques for deep neural networks in a dynamic environment. (2018).
[54] Stuart Geman, Elie Bienenstock, and René Doursat. 1992. Neural networks and the bias/variance dilemma. *Neural Comput.* 4, 1 (1992), 1–58.
[55] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the ACM Symposium on Operating Systems Principles*.
[56] Andrew Gibiansky. 2017. Bringing HPC Techniques to Deep Learning. Retrieved from http://research.baidu.com/bringing-hpc-techniques-deep-learning/.
[57] Yue-Jiao Gong, Wei-Neng Chen, Zhi-Hui Zhan, Jun Zhang, Yun Li, Qingfu Zhang, and Jing-Jing Li. 2015. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Appl. Soft Comput.* 34 (2015), 286–300. DOI: https://doi.org/10.1016/j.asoc.2015.04.061
[58] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems, Vol. 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2672–2680.
[59] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. 2011. Sorting, searching, and simulation in the MapReduce framework. In *Proceedings of the International Symposium on Algorithms and Computation*. 374–383.
[60] Google. 2017. Google Cloud TPU. Retrieved from https://cloud.google.com/tpu.
[61] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *CoRR* abs/1706.02677 (2017). arxiv:1706.02677 http://arxiv.org/abs/1706.02677.
[62] William D. Gropp, William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: Portable Parallel Programming with the Message-passing Interface*. Vol. 1. The MIT Press.
[63] Mehmet Gönen. 2012. Predicting drug-target interactions from chemical and genomic kernels using Bayesian matrix factorization. *Bioinformatics* 28, 18 (2012), 2304–2310.
[64] D. Hall, D. Ramage, et al. 2009. Breeze: Numerical Processing Library for Scala. Retrieved from https://github. com/scalanlp/breeze.
[65] Minyang Han and Khuzaima Daudjee. 2015. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proc. VLDB Endow.* 8, 9 (May 2015), 950–961. DOI: https://doi.org/10.14778/2777598.2777604
[66] Tianyi David Han and Tarek S. Abdelrahman. 2011. Reducing branch divergence in GPU programs. In *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units*. 3 (Mar. 2011) 1–8.
[67] Elmar Haußmann. 2018. Accelerating I/O Bound Deep Learning on Shared Storage. Retrieved from https://blog.riseml.com/accelerating-io-bound-deep-learning-e0e3f095fd0.
[68] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *CoRR* abs/1512.03385 (2015). arxiv:1512.03385 http://arxiv.org/abs/1512.03385.
[69] Magnus Rudolph Hestenes, Eduard Stiefel, and others. 1952. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards* 49, 6 (1952), 409–436.
[70] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580* (2012).
[71] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. 2017. Deep models under the GAN: Information leakage from collaborative deep learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 603–618.
[72] Thomas Hofmann. 1999. Probabilistic latent semantic analysis. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann Publishers Inc., 289–296.
[73] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. 2017. Gaia: Geo-distributed machine learning approaching LAN speeds. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, 629–647. Retrieved from https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/hsieh.
[74] IBM Cloud. 2018. IBM Watson Machine Learning. Retrieved from https://www.ibm.com/cloud/machine-learning.
[75] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 59–72.

[76] Sylvain Jeaugey. 2017. NCCL 2.0. Retrieved from http://on-demand.gputechconf.com/gtc/2017/presentation/s7155-jeaugey-nccl.pdf.
[77] Genlin Ji and Xiaohan Ling. 2007. Ensemble learning based distributed clustering. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 312–321.
[78] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*. ACM, 675–678.
[79] Michael I. Jordan and Tom M. Mitchell. 2015. Machine learning: Trends, perspectives, and prospects. *Science* 349, 6245 (2015), 255–260.
[80] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the ACM/IEEE 44th International Symposium on Computer Architecture (ISCA'17)*. IEEE, 1–12.
[81] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement learning: A survey. *J. Artific. Intell. Res.* 4 (1996), 237–285.
[82] Amir E. Khandani, Adlar J. Kim, and Andrew W. Lo. 2010. Consumer credit-risk models via machine-learning algorithms. *J. Bank. Fin.* 34, 11 (2010), 2767–2787.
[83] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (Aug. 2009), 30–37.
[84] Thorsten Kurth, Jian Zhang, Nadathur Satish, Evan Racah, Ioannis Mitliagkas, Md Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, et al. 2017. Deep learning at 15pf: Supervised and semi-supervised classification for scientific data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 7.
[85] Donghwoon Kwon, Hyunjoo Kim, Jinoh Kim, Sang C. Suh, Ikkyun Kim, and Kuinam J. Kim. 2017. A survey of deep learning-based network anomaly detection. *Cluster Comput.* (27 Sep. 2017). DOI: https://doi.org/10.1007/s10586-017-1117-8
[86] Ralf Lammel. 2008. Google's MapReduce programming model—Revisited. *Sci. Comput. Prog.* 70, 1 (2008), 1.
[87] Sara Landset, Taghi M. Khoshgoftaar, Aaron N. Richter, and Tawfiq Hasanin. 2015. A survey of open source tools for machine learning with big data in the Hadoop ecosystem. *J. Big Data* 2, 1 (2015), 24.
[88] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Y. Bengio. 2007. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the International Conference on Machine Learning*, Vol. 227, 473–480. DOI: https://doi.org/10.1145/1273496.1273556
[89] Chuck L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.* 5, 3 (1979), 308–323.
[90] Quoc V. Le. 2013. Building high-level features using large scale unsupervised learning. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'13)*. IEEE, 8595–8598.
[91] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. 2017. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 8.
[92] Mu Li, David G. Andersen, Alexander Smola, and Kai Yu. 2014. Communication efficient distributed machine learning with the parameter server. In *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS'14)*, Vol. 1. The MIT Press, 19–27.
[93] Mu Li, David G. Anderson, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 583–598.
[94] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G. Andersen, and Alexander Smola. 2013. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, Vol. 6. 2 pages.
[95] Dong C. Liu and Jorge Nocedal. 1989. On the limited memory BFGS method for large scale optimization. *Math. Prog.* 45, 1–3 (1989), 503–528.
[96] A. R. Mamidala, G. Kollias, C. Ward, and F. Artico. 2018. MXNET-MPI: Embedding MPI parallelism in parameter server task model for scaling deep learning. *ArXiv e-prints* (Jan. 2018). arxiv:cs.DC/1801.03855.
[97] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. 2016. Federated learning of deep networks using model averaging. (2016).
[98] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, et al. 2016. MLlib: Machine learning in Apache Spark. *J. Mach. Learn. Res.* 17, 1 (2016), 1235–1241.

[99] Cade Metz. 2018. Big bets on AI open a new frontier for chip start-ups, too. *The New York Times* 14 Jan. (2018). Retrieved from https://www.nytimes.com/2018/01/14/technology/artificial-intelligence-chip-start-ups.html.
[100] Ryan J. Meuth. 2007. GPUs surpass computers at repetitive calculations. *IEEE Potent.* 26, 6 (2007), 12–23.
[101] Microsoft. 2018. Microsoft Azure Machine Learning. Retrieved from https://azure.microsoft.com/en-us/overview/machine-learning/.
[102] Microsoft Inc. 2015. Distributed Machine Learning Toolkit (DMTK). Retrieved from http://www.dmtk.io.
[103] Jyoti Nandimath, Ekata Banerjee, Ankur Patil, Pratima Kakade, Saumitra Vaidya, and Divyansh Chaturvedi. 2013. Big data analysis using Apache Hadoop. In *Proceedings of the IEEE 14th International Conference on Information Reuse & Integration (IRI'13)*. IEEE, 700–703.
[104] Fairuz Amalina Narudin, Ali Feizollah, Nor Badrul Anuar, and Abdullah Gani. 2016. Evaluation of machine learning classifiers for mobile malware detection. *Soft Comput.* 20, 1 (2016), 343–357.
[105] David Newman, Arthur Asuncion, Padhraic Smyth, and Max Welling. 2009. Distributed algorithms for topic models. *J. Mach. Learn. Res.* 10, Aug. (2009), 1801–1828.
[106] NVIDIA Corporation. 2015. NVIDIA Collective Communications Library (NCCL). Retrieved from https://developer.nvidia.com/nccl.
[107] NVIDIA Corporation. 2017. Nvidia Tesla V100. Retrieved from https://www.nvidia.com/en-us/data-center/tesla-v100/.
[108] Kyoung-Su Oh and Keechul Jung. 2004. GPU implementation of neural networks. *Pattern Recog.* 37, 6 (2004), 1311–1314.
[109] Andreas Olofsson. 2016. Epiphany-V: A 1024 processor 64-bit RISC system-on-chip. (2016).
[110] Andreas Olofsson, Tomas Nordström, and Zain Ul-Abdin. 2014. Kickstarting high-performance energy-efficient manycore architectures with epiphany. *arXiv preprint arXiv:1412.5538* (2014).
[111] David W. Opitz and Richard Maclin. 1999. Popular ensemble methods: An empirical study. (1999).
[112] Róbert Ormándi, István Hegedűs, and Márk Jelasity. 2013. Gossip learning with linear models on fully distributed data. *Concurr. Comput.: Pract. Exper.* 25, 4 (2013), 556–571.
[113] A. Orriols-Puig, J. Casillas, and E. Bernado-Mansilla. 2009. Fuzzy-UCS: A Michigan-style learning fuzzy-classifier system for supervised learning. *IEEE Trans. Evol. Comput.* 13, 2 (Apr. 2009), 260–283.
[114] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S. Chung. 2015. Accelerating deep convolutional neural networks using specialized hardware. *Micros. Res. Whitep.* 2, 11 (2015).
[115] Matthew Felice Pace. 2012. BSP vs MapReduce. *Proced. Comput. Sci.* 9 (2012), 246–255.
[116] Louis Papageorgiou, Picasi Eleni, Sofia Raftopoulou, Meropi Mantaiou, Vasileios Megalooikonomou, and Dimitrios Vlachakis. 2018. Genomic big data hitting the storage bottleneck. *EMBnet. J.* 24 (2018).
[117] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).
[118] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.* 69, 2 (Feb. 2009), 117–124.
[119] Diego Peteiro-Barral and Bertha Guijarro-Berdiñas. 2013. A survey of methods for distributed machine learning. *Prog. Artific. Intell.* 2, 1 (2013), 1–11.
[120] Boris T. Polyak. 2007. Newton's method and its use in optimization. *Europ. J. Operat. Res.* 181, 3 (2007), 1086–1096.
[121] Daniel Pop. 2016. Machine learning and cloud computing: Survey of distributed and SaaS solutions. *arXiv preprint arXiv:1603.08767* (2016).
[122] Foster Provost and Venkateswarlu Kolluri. 1999. A survey of methods for scaling up inductive algorithms. *Data Min. Knowl. Disc.* 3, 2 (1999), 131–169.
[123] Junfei Qiu, Qihui Wu, Guoru Ding, Yuhua Xu, and Shuo Feng. 2016. A survey of machine learning for big data processing. *EURASIP J. Adv. Sig. Proc.* 2016, 1 (2016), 67.
[124] Ioan Raicu, Ian Foster, Alex Szalay, and Gabriela Turcu. 2006. Astroportal: A science gateway for large-scale astronomy data analysis. In *Proceedings of the TeraGrid Conference*. 12–15.
[125] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. 2009. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th International Conference on Machine Learning*. ACM, 873–880.
[126] James Reinders. 2013. AVX-512 instructions. *Intel Corporation* (2013).
[127] Peter Richtárik and Martin Takáč. 2016. Distributed coordinate descent method for learning with big data. *J. Mach. Learn. Res.* 17, 1 (2016), 2657–2681.
[128] Kaz Sato, Cliff Young, and David Patterson. 2017. An in-depth look at Google's first Tensor Processing Unit (TPU). *Google Cloud Big Data Mach. Learn. Blog* 12 (2017).
[129] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2135–2135.

[130] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech DNNs. In *Proceedings of the Conference of the International Speech Communication Association (Interspeech'14)*. Retrieved from https://www.microsoft.com/en-us/research/publication/1-bit-stochastic-gradient-descent-and-application-to-data-parallel-distributed-training-of-speech-dnns/.

[131] Alexander Sergeev and Mike Del Balso. 2018. Horovod: Fast and easy distributed deep learning in TensorFlow. Retrieved from https://arxiv.org/abs/1802.05799.

[132] Pierre Sermanet, Soumith Chintala, and Yann LeCun. 2012. Convolutional neural networks applied to house numbers digit classification. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR'12)*. IEEE, 3288–3291.

[133] Pierre Sermanet and Yann LeCun. 2011. Traffic sign recognition with multi-scale convolutional networks. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'11)*. IEEE, 2809–2813.

[134] Amazon Web Services. 2016. Introducing Amazon EC2 P2 Instances, the Largest GPU-Powered Virtual Machine in the Cloud. Retrieved from https://aws.amazon.com/about-aws/whats-new/2016/09/introducing-amazon-ec2-p2-instances-the-largest-gpu-powered-virtual-machine-in-the-cloud/.

[135] Amazon Web Services. 2017. Amazon EC2 F1 Instances. Retrieved from https://aws.amazon.com/ec2/instance-types/f1/.

[136] Shai Shalev-Shwartz and Tong Zhang. 2013. Stochastic dual coordinate ascent methods for regularized loss minimization. (2013).

[137] James G. Shanahan and Laing Dai. 2015. Large scale distributed data science using Apache Spark. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2323–2324.

[138] Shaohuai Shi and Xiaowen Chu. 2017. Performance modeling and evaluation of distributed deep learning frameworks on GPUs. *CoRR* abs/1711.05979 (2017). arxiv:1711.05979 Retrieved from http://arxiv.org/abs/1711.05979.

[139] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. 2016. Benchmarking state-of-the-art deep learning software tools. In *Proceedings of the 7th International Conference on Cloud Computing and Big Data (CCBD'16)*. IEEE, 99–104.

[140] Reza Shokri and Vitaly Shmatikov. 2015. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1310–1321.

[141] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop distributed file system. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. IEEE, 1–10.

[142] J. Simm, A. Arany, P. Zakeri, T. Haber, J. K. Wegner, V. Chupakhin, H. Ceulemans, and Y. Moreau. 2017. Macau: Scalable Bayesian factorization with high-dimensional side information using MCMC. In *Proceedings of the IEEE 27th International Workshop on Machine Learning for Signal Processing (MLSP'17)*. DOI: https://doi.org/10.1109/MLSP.2017.8168143

[143] Dilpreet Singh and Chandan K. Reddy. 2015. A survey on platforms for big data analytics. *J. Big Data* 2, 1 (2015), 8.

[144] Michael John Sebastian Smith. 1997. *Application-specific Integrated Circuits*. Vol. 7. Addison-Wesley, Reading, MA.

[145] Alexander Smola and Shravan Narayanamurthy. 2010. An architecture for parallel topic models. *Proc. VLDB Endow.* 3, 1–2 (2010), 703–710.

[146] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian optimization of machine learning algorithms. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*, Vol. 25, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2951–2959.

[147] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going deeper with convolutions. *CoRR* abs/1409.4842 (2014). arxiv:1409.4842 http://arxiv.org/abs/1409.4842.

[148] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2015. Rethinking the inception architecture for computer vision. *CoRR* abs/1512.00567 (2015). arxiv:1512.00567 http://arxiv.org/abs/1512.00567.

[149] Martin Takác, Avleen Singh Bijral, Peter Richtárik, and Nati Srebro. 2013. Mini-batch primal and dual methods for SVMs. In *Proceedings of the International Conference on Machine Learning (ICML'13)*. 1022–1030.

[150] The Khronos Group. 2018. Neural Network Exchange Format (NNEF). Retrieved from https://www.khronos.org/registry/NNEF/specs/1.0/nnef-1.0.pdf.

[151] K. I. Tsianos, S. F. Lawlor, and M. G. Rabbat. 2012. Communication/computation tradeoffs in consensus-based distributed optimization. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*.

[152] Sujatha R. Upadhyaya. 2013. Parallel approaches to machine learning—A comprehensive survey. *J. Parallel Distrib. Comput.* 73, 3 (2013), 284–292.

[153] Praneeth Vepakomma, Tristan Swedish, Ramesh Raskar, Otkrist Gupta, and Abhimanyu Dubey. 2018. No peek: A survey of private distributed deep learning. *arXiv preprint arXiv:1812.03288* (2018).

[154] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. 2012. AIOLOS: Middleware for improving mobile application performance through cyber foraging. *J. Syst. Softw.* 85, 11 (2012), 2629–2639.
[155] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2015. Managed communication and consistency for fast data-parallel iterative analytics. (2015), 381–394.
[156] Sholom M. Weiss and Nitin Indurkhya. 1995. Rule-based machine learning methods for functional prediction. *J. Artific. Intell. Res.* 3 (1995), 383–403.
[157] Stewart W. Wilson. 1995. Classifier fitness based on accuracy. *Evolut. Comput.* 3, 2 (1995), 149–175.
[158] Stephen J. Wright. 2015. Coordinate descent algorithms. *Math. Prog.* 151, 1 (2015), 3–34.
[159] P. Xie, J. K. Kim, Y. Zhou, Q. Ho, A. Kumar, Y. Yu, and E. Xing. 2015. Distributed machine learning via sufficient factor broadcasting. (2015).
[160] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. 2013. Petuum: A new platform for distributed machine learning on big data. *ArXiv e-prints* (Dec. 2013). arxiv:stat.ML/1312.7651.
[161] Eric P. Xing, Qirong Ho, Pengtao Xie, and Dai Wei. 2016. Strategies and principles of distributed machine learning on big data. *Engineering* 2, 2 (2016), 179–195.
[162] Feng Yan, Olatunji Ruwase, Yuxiong He, and Evgenia Smirni. 2016. SERF: Efficient scheduling for fast deep neural network serving via judicious parallelism. In *Proceedings of the International Conference forHigh Performance Computing, Networking, Storage and Analysis (SC'16)*. IEEE, 300–311.
[163] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. 2016. TR-Spark: Transient computing for big data analytics. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*. ACM, New York, NY, 484–496.
[164] Dani Yogatama, Phil Blunsom, Chris Dyer, Edward Grefenstette, and Wang Ling. 2016. Learning to compose words into sentences with reinforcement learning. *CoRR* abs/1611.09100 (2016). arxiv:1611.09100 http://arxiv.org/abs/1611.09100.
[165] Yang You, Aydın Buluç, and James Demmel. 2017. Scaling deep learning on GPU and Knights Landing clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM.
[166] Haifeng Yu and Amin Vahdat. 2002. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.* 20, 3 (2002), 239–282.
[167] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.
[168] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)* 10, 10–10 (2010), 95.
[169] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, et al. 2016. Apache Spark: A unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.
[170] Li Zeng, Ling Li, Lian Duan, Kevin Lu, Zhongzhi Shi, Maoguang Wang, Wenjuan Wu, and Ping Luo. 2012. Distributed data mining: A survey. *Inf. Technol. Manag.* 13, 4 (2012), 403–409.
[171] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. *arXiv preprint* (2017).

Received December 2018; revised November 2019; accepted December 2019