Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis

TAL BEN-NUN* and TORSTEN HOEFLER, ETH Zurich

Deep Neural Networks (DNNs) are becoming an important tool in modern computing applications. Accelerating their training is a major challenge and techniques range from distributed algorithms to low-level circuit design. In this survey, we describe the problem from a theoretical perspective, followed by approaches for its parallelization. Specifically, we present trends in DNN architectures and the resulting implications on parallelization strategies. We discuss the different types of concurrency in DNNs; synchronous and asynchronous stochastic gradient descent; distributed system architectures; communication schemes; and performance modeling. Based on these approaches, we extrapolate potential directions for parallelism in deep learning.

CCS Concepts: • **General and reference** → *Surveys and overviews*; • **Computing methodologies** → **Neural networks**; **Distributed computing methodologies**; **Parallel computing methodologies**; *Machine learning*;

Additional Key Words and Phrases: Deep Learning, Distributed Computing, Parallel Algorithms

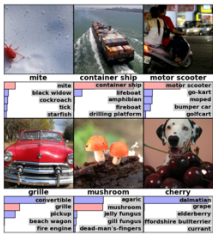
ACM Reference format:
Tal Ben-Nun and Torsten Hoefler. 2018. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. 60 pages.

1 INTRODUCTION

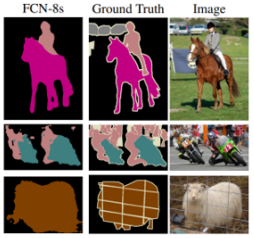
Machine Learning, and in particular Deep Learning [LeCun et al. 2015], is a field that is rapidly taking over a variety of aspects in our daily lives. In the core of deep learning lies the Deep Neural Network (DNN), a construct inspired by the interconnected nature of the human brain. Trained properly, the expressiveness of DNNs provides accurate solutions for problems previously thought to be unsolvable, simply by observing large amounts of data. Deep learning has been successfully implemented for a plethora of subjects, ranging from image classification [Huang et al. 2017], through speech recognition [Amodei et al. 2016] and medical diagnosis [Cireşan et al. 2013], to autonomous driving [Bojarski et al. 2016] and defeating human players in complex games [Silver et al. 2017] (see Fig. 1 for more examples).

Since the 1980s, neural networks have attracted the attention of the machine learning community [LeCun et al. 1989]. However, DNNs only rose into prominence once the available computational power permitted training on workstations by exploiting their inherent parallelism. Consequently, the first successful modern DNN, AlexNet [Krizhevsky et al. 2012], managed to outperform existing

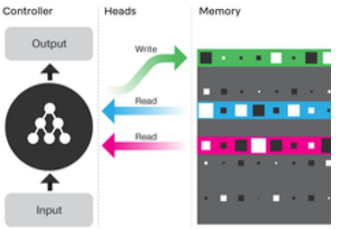
*The corresponding author



(a) Object Classification [Krizhevsky et al. 2012]



(b) Image Segmentation [Long et al. 2015]



(c) Neural Computers [Graves et al. 2016]

Fig. 1. Deep Learning Usage Examples (figures adapted from respective citations)

classification methods by a factor of two, and current prevalent DNNs [Huang et al. 2017] outperform AlexNet by an additional factor of ~2.9.

As datasets increase in size and DNNs in complexity, the computational intensity and memory demands of deep learning increase proportionally. Training a DNN to competitive accuracy today essentially requires a cluster of machines with high-performance computing architectures. To harness the computational power available in such systems, different aspects of training and inference (evaluation) of DNNs have been modified to increase their underlying concurrency.

In this survey, we discuss the variety of topics in the context of parallelism and distribution, spanning from vectorization to efficient use of supercomputers. In particular, we present parallelism strategies for DNN evaluation and implementations thereof, as well as extensions to training algorithms and systems targeted at supporting distributed environments. To provide comparative measures on the approaches, we analyze their concurrency and average parallelism using the Work-Depth model [Blumofe and Leiserson 1999].

1.1 Related Surveys

Other surveys in the field of deep learning focus on applications of deep learning [Najafabadi et al. 2015], neural networks and their history [LeCun et al. 2015; Li 2017; Schmidhuber 2015; Wang et al. 2017], scaling up deep learning [Bengio 2013], and hardware architectures for DNNs [Ienne 1993; Lacey et al. 2016; Sze et al. 2017].

As for neural networks, three surveys [LeCun et al. 2015; Schmidhuber 2015; Wang et al. 2017] describe DNNs and the origins of deep learning methodologies from a historical perspective, as well as discuss the potential capabilities of DNNs w.r.t. learnable functions and representational power. Two of the three surveys [Schmidhuber 2015; Wang et al. 2017] also describe optimization methods and applied regularization techniques in detail.

Another paper [Bengio 2013] discusses scaling deep learning from various perspectives, focusing on models, optimization algorithms, and datasets. The paper also overviews some aspects of distributed computing, including asynchronous and sparse communication.

Surveys of hardware architectures mostly focus on the computational side of training rather than the optimization. This includes a recent survey [Sze et al. 2017] that reviews computation techniques for DNN layers and mapping computations to hardware, exploiting inherent parallelism. The survey also includes discussion on data representation reduction (e.g., via quantization) to reduce overall memory bandwidth within the hardware. Other surveys discuss accelerators for traditional neural networks [Ienne 1993] and the use of FPGAs in deep learning [Lacey et al. 2016].

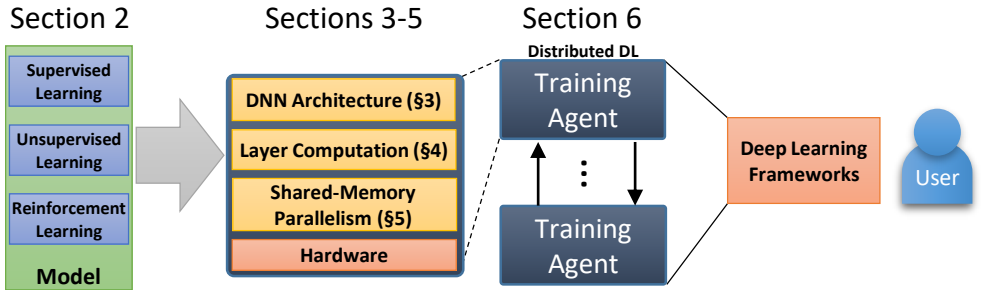


Fig. 2. Concurrent DNN Training Infrastructure

1.2 Scope

In this paper, we provide a comprehensive survey and analysis of parallel and distributed deep learning, summarized in Fig. 2 and organized as follows:

- Section 2 defines the terminology and algorithms used with Deep Neural Networks (DNNs), as well as terminology in parallel computer architecture and algorithms.
- Section 3 presents DNNs in detail, elaborating on the different layers, analyzing inherent concurrency, and presenting computational trends in popular DNNs over the years.
- Section 4 describes how layers presented in Section 3 can be computed differently to expose more concurrency at the cost of redundant computations.
- Section 5 explores and analyzes the main approaches for shared-memory parallelism with respect to DNNs and their training.
- Section 6 provides an overview of distributed-memory DNN training, describing modifications to training algorithms, techniques to reduce communication, and system implementations.
- Section 7 gives concluding remarks and extrapolates potential directions in the field.

The paper surveys 227 other works, obtained by recursively tracking relevant bibliography from seminal papers in the field, dating back to the year 1984. We include additional papers resulting from keyword searches on Google Scholar[1] and arXiv[2]. Due to the quadratic increase in deep learning papers on the latter source (Table 1), some works may have not been included. The full list of categorized papers in this survey can be found online[3].

Table 1. Yearly arXiv Papers in Computer Science (AI and Computer Vision)

Year	2012	2013	2014	2015	2016	2017
cs.AI	1,081	1,765	1,022	1,105	1,929	2,790
cs.CV	577	852	1,349	2,261	3,627	5,693

2 TERMINOLOGY AND ALGORITHMS

This section establishes theory and naming conventions for the material presented in the survey. We first discuss the classes of supervised, unsupervised, and reinforcement learning problems, followed by relevant foundations of parallel programming.

[1] https://scholar.google.com/
[2] https://www.arxiv.org/
[3] https://spcl.inf.ethz.ch/Research/Parallel_Programming/DistDL/

Name	Definition
$\mathcal{D}$	Data probability distribution
S	Training dataset
$w \in \mathcal{H}$	Model parameters. $w_i^{(t)}$ denotes parameter i at SGD iteration t
$f_w(z)$	Model function (learned predictor)
$h(z)$	Ground-truth label (in Supervised Learning)
$\ell(w, z)$	Per-sample loss function
$\nabla\ell(w, z)$	Gradient of ℓ
$u(g, w, t)$	Parameter update rule. Function of loss gradient g, parameters w, and iteration t

Table 2. Summary of Notations

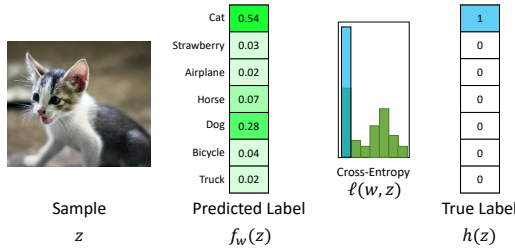


Fig. 3. Multi-Class Classification Loss

2.1 Supervised Learning

In machine learning, Supervised Learning [Shalev-Shwartz and Ben-David 2014] is the process of optimizing a function from a set of labeled samples (*dataset*) such that, given a sample, the function would return a value that approximates the label. It is assumed that both the dataset and other, unobserved samples, are sampled from the same probability distribution.

Throughout the survey, we refer to the operators $\mathbb{P}$ and $\mathbb{E}$ as the probability and expectation of random variables; $z \sim \mathcal{D}$ denotes that a random variable z is sampled from a probability distribution $\mathcal{D}$; and $\mathbb{E}_{z\sim\mathcal{D}}[f(z)]$ denotes the expected value of $f(z)$ for a random variable z. The notations used in this section are summarized in Table 2.

Formally, given a probability distribution of data $\mathcal{D}$, random variable $z \sim \mathcal{D}$, a domain X where we construct samples from, a label domain Y, and a hypothesis class $\mathcal{H}$ containing functions $f : X \to Y$, we wish to minimize the *loss function* $L_{\mathcal{D}}(f) \equiv \mathbb{P}[f(z) \neq h(z)]$ (sometimes called the *generalization error*), where $h(z)$ represents the true label of z. In practice, it is common to use a class $\mathcal{H}$ of functions f_w that are defined by a vector of parameters w (also denoted as θ), in order to define a parameter space. For example, $\mathcal{H}$ may represent an N-dimensional hyperplane that separates between samples of two classes, where w_i are its coefficients. In deep neural networks, we define w in multiple layers, namely, $w_{l,i}$ is the parameter at layer l and index i.

We wish to find w^* that minimizes the above loss function, as follows:

$$w^* = \arg\min_{w\in\mathcal{H}} L_{\mathcal{D}}(f_w) = \arg\min_{w\in\mathcal{H}} \mathop{\mathbb{E}}_{z\sim\mathcal{D}} [\ell(w, z)], \tag{1}$$

where $\ell : \mathcal{H} \times X \to \mathbb{R}_+$ is the loss of an individual sample.

In this work, we consider two types of supervised learning problems, from which the sample loss functions are derived: *(multi-class) classification* and *regression*. In the former, the goal is to identify which class a sample most likely belongs to, e.g., inferring what type of animal appears in an image. In the latter type, the goal is to find a relation between the domains X and Y, predicting values in Y for new samples in X. For instance, a regression problem might predict the future temperature of a region, given past observations.

For minimization purposes, a sample loss function ℓ should be continuous and differentiable. For regression problems, it is possible to use loss functions such as the squared difference $\ell(w, z) = (f_w(z) - h(z))^2$. On the other hand, in classification problems, the simplest definition of loss is the binary (0–1) loss, i.e., $\ell(w, z) = 0$ if $f_w(z) = h(z)$ or 1 otherwise. However, this definition does not match the continuity and differentiability criteria.

To resolve this issue, prominent multi-class classification problems define Y as a probability distribution of the inferred class types (see Fig. 3), instead of a single label. The loss function then computes the difference of the predicted distribution from the true label "distribution". To normalize the model output, the *softmax* function $\sigma(z)_i = \frac{\exp(z_i)}{\sum_k \exp(z_k)}$ is typically used, and the difference between distributions is achieved using cross-entropy: $\ell(w, z) = -\sum_i h(z)_i \log \sigma\left(f_w(z)\right)_i$. The cross-entropy loss can be seen as a generalization of logistic regression, which introduces a continuous loss function for multi-class classification.

Minimizing the loss function can be performed by using different approaches, such as iterative methods (e.g., BFGS [Nocedal and Wright 2006]) or meta-heuristics (e.g., evolutionary algorithms [Real et al. 2018]). A prominent method in machine learning is iterative Gradient Descent. However, since the full $\mathcal{D}$ is never observed, it is necessary to obtain an unbiased estimator of the gradient. Observe that $\nabla L_{\mathcal{D}}(w) = \mathbb{E}_{z\sim\mathcal{D}}\left[\nabla\ell(w, z)\right]$ (Eq. 1, linearity of the derivative). Thus, in expectation, we can perform gradient descent on randomly sampled data in each iteration, applying *Stochastic Gradient Descent* (SGD).

The SGD algorithm iteratively optimizes parameters defined by the sequence $\{w^{(t)}\}_{t=0}^{T}$, using samples from a dataset S that is assumed to be sampled from $\mathcal{D}$. SGD is proven to converge with a rate of $O(1/\sqrt{T})$ for convex functions with Lipschitz-continuous and bounded gradient [Nemirovski et al. 2009]. The algorithm is defined as follows:

Algorithm 1 Stochastic Gradient Descent (SGD)

1: **for** $t = 0$ **to** T **do** ▷ Stopping condition
2: $z \leftarrow$ Random element from S ▷ Sample dataset S
3: $g \leftarrow \nabla\ell(w^{(t)}, z)$ ▷ Compute gradient of ℓ
4: $w^{(t+1)} \leftarrow w^{(t)} + u(g, w^{(0,\ldots,t)}, t)$ ▷ Update weights with function u
5: **end for**

Prior to running SGD, the program must choose an initial estimate for the weights $w^{(0)}$. Due to the ill-posed nature of some machine learning problems, the selection of $w^{(0)}$ is important and may reflect on the final loss value. The choice of initial weights can originate from random values, informed decisions (e.g., Xavier randomized initialization [Glorot and Bengio 2010]), or from pre-trained weights (or a subset thereof) in a methodology called *Transfer Learning* [Pan and Yang 2010]. In deep learning, recent works state that the optimization space is riddled with saddle points [LeCun et al. 2015], and assume that the value of $w^{(0)}$ does not affect the final loss. In practice, however, improper initialization may have an adverse effect on loss values as networks become deeper [He et al. 2015].

In line 1, T denotes the number of steps to run SGD for (known as the *stopping condition* or *computational budget*). Typically, real-world instances of SGD run for a constant number of steps, for a fixed period of time, or until a desired accuracy is achieved. Line 2 then samples random elements from the dataset, so as to provide the unbiased loss estimator. In practice, this is implemented by shuffling the dataset S before the loop, and processing that permutation entirely in a step called *epoch*. A full training procedure usually consists of tens to hundreds of such epochs [Goyal et al. 2017; You et al. 2017c].

After an element is sampled, the gradient of the loss function with respect to the weights $w^{(t)}$ is computed (line 3). In deep neural networks, the gradient is obtained with respect to each layer ($w_l^{(t)}$) using backpropagation (see Section 3.2). The obtained gradient is then used for updating the current weights, using a *weight update rule* (line 4).

Table 3. Popular Weight Update Rules

Method	Formula
Learning Rate	$w^{(t+1)} = w^{(t)} - \eta \cdot \nabla \ell(w^{(t)}, z) \quad = w^{(t)} - \eta \cdot \nabla w^{(t)}$
Adaptive Learning Rate	$w^{(t+1)} = w^{(t)} - \eta_t \cdot \nabla w^{(t)}$
Momentum [Qian 1999]	$w^{(t+1)} = w^{(t)} + \mu \cdot (w^{(t)} - w^{(t-1)}) - \eta \cdot \nabla w^{(t)}$
Nesterov Momentum [Nesterov 1983]	$w^{(t+1)} = w^{(t)} + v_t; \quad v_{t+1} = \mu \cdot v_t - \eta \cdot \nabla \ell(w^{(t)} - \mu \cdot v_t, z)$
AdaGrad [Duchi et al. 2011]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot \nabla w_i^{(t)}}{\sqrt{A_{i,t}} + \varepsilon}; \quad A_{i,t} = \sum_{\tau=0}^{t} \left(\nabla w_i^{(t)}\right)^2$
RMSProp [Hinton 2012]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot \nabla w_i^{(t)}}{\sqrt{A'_{i,t}} + \varepsilon}; \quad A'_{i,t} = \beta \cdot A'_{t-1} + (1-\beta)\left(\nabla w_i^{(t)}\right)^2$
Adam [Kingma and Ba 2015]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot M_{i,t}^{(1)}}{\sqrt{M_{i,t}^{(2)}} + \varepsilon}; \quad M_{i,t}^{(m)} = \frac{\beta_m \cdot M_{i,t-1}^{(m)} + (1-\beta_m)\left(\nabla w_i^{(t)}\right)^m}{1-\beta_m^t}$

2.1.1 Weight Update Rules. The weight update rule, denoted as u in the algorithm, can be defined as a function of the gradient g, the previous weight values $w^{(0)}, \cdots, w^{(t)}$, and the current iteration t. Table 3 summarizes the popular u functions used in training. In the table, the basic SGD update rule is $u_{sgd}(g) = -\eta \cdot g$, where η represents the *learning rate*. η controls how much the gradient values will overall affect the next estimate $w^{(t+1)}$, and in iterative nonlinear optimization methods finding the correct η is a considerable part of the computation [Nocedal and Wright 2006]. In machine learning problems, it is customary to fix η, or set an iteration-based weight update rule $u_{alr}(g, t) = -\eta_t \cdot g$, where η_t decreases (decays) over time to bound the modification size and avoid local divergence.

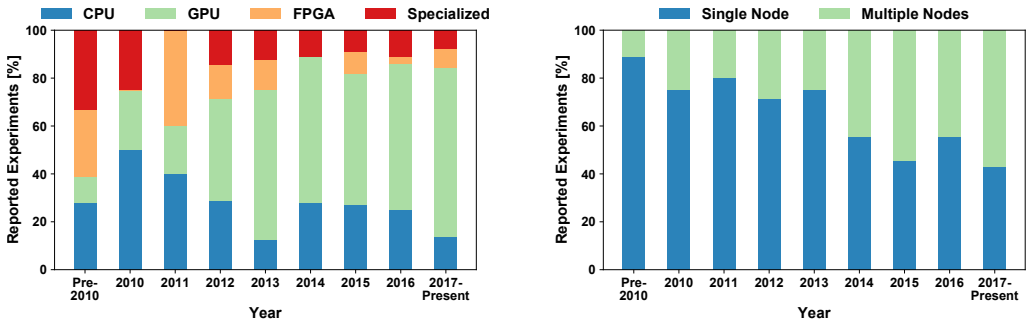
Other popular weight update rules include *Momentum*, which uses the difference between current and previous weights $w^{(t)} - w^{(t-1)}$ to overcome local minima with natural motion [Nesterov 1983; Qian 1999]. More recent update rules, such as RMSProp [Hinton 2012] and Adam [Kingma and Ba 2015], use the first and second moments of the gradient in order to adapt the learning rate per-weight (as seen in the table), enhancing sparser updates over others.

Factors such as the learning rate and other symbols found in Table 3 are called *hyper-parameters*, and are set before the optimization process begins. In the table, μ, β, β_1, and β_2 represent the momentum, RMS decay rate, and first and second moment decay rate hyper-parameters, respectively. To obtain the best results, hyper-parameters must be tuned, which can be performed by value sweeps or by meta-optimization [Jaderberg et al. 2017]. The multitude of hyper-parameters and the reliance upon them is considered problematic by a part of the community [Rahimi and Recht 2017].

2.2 Unsupervised and Reinforcement Learning

Two other classes in machine learning are *unsupervised* and *reinforcement learning*. In the first class, the dataset S is not labeled (i.e., $h(z)$ does not exist) and training typically results in different objective functions, intended to infer structure from the unlabeled data. The second class refers to tasks where an environment is observed at given points in time, and training optimizes an action policy function to maximize the reward of the observer.

In the context of deep learning, unsupervised learning has two useful applications: auto-encoders, and Generative Adversarial Networks (GANs) [Goodfellow et al. 2014]. Auto-encoders can be constructed as neural networks that receive a sample x as input, and output a value as close to x as possible. When training such networks, it is possible to, for instance, feed samples with artificially-added noise and optimize the network to return the original sample (e.g., using a squared



(a) Hardware Architectures

(b) Training with Single vs. Multiple Nodes

Fig. 4. Parallel Architectures in Deep Learning

loss function), in order to learn de-noising filters. Alternatively, similar techniques can be used to learn image compression networks [Ballé et al. 2017].

GANs [Goodfellow et al. 2014] are a recent development in machine learning. They employ deep neural networks to generate realistic data (typically images) by simultaneously training two networks. The first (discriminator) network is trained to distinguish "real" dataset samples from "fake" generated samples, while the second (generator) network is trained to generate samples that are as similar to the dataset as possible.

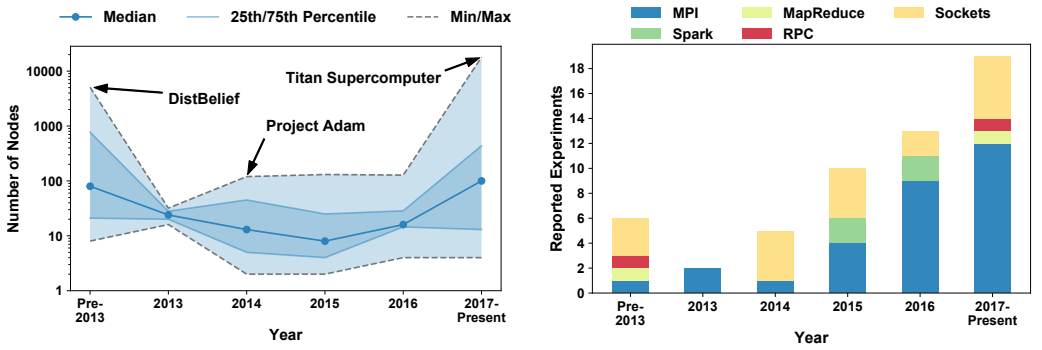
The class of Reinforcement Learning (RL) utilizes DNNs [Li 2017] for different purposes, such as defining policy functions and reward functions. Training algorithms for RL differ from supervised and unsupervised learning, using methods such as Deep Q Learning [Mnih et al. 2015] and A3C [Mnih et al. 2016]. The details of these algorithms are out of the scope of this survey, but their parallelization techniques are similar.

2.3 Parallel Computer Architecture

We continue with a brief overview of parallel hardware architectures that are used to execute learning problems in practice. They can be roughly classified into single-machine (often shared memory) and multi-machine (often distributed memory) systems.

2.3.1 Single-machine Parallelism. Parallelism is ubiquitous in today's computer architecture, internally on the chip in the form of pipelining and out-of-order execution as well as exposed to the programmer in the form of multi-core or multi-socket systems. Multi-core systems have a long tradition and can be programmed with either multiple processes (different memory domains) or multiple threads (shared memory domains) or a mix of both. The main difference is that multi-process parallel programming forces the programmer to consider the distribution of the data as a first-class concern while multi-threaded programming allows the programmer to only reason about the parallelism, leaving the data shuffling to the hardware system (often through hardware cache-coherence protocols).

General-purpose CPUs have been optimized for general workloads ranging from event-driven desktop applications to datacenter server tasks (e.g., serving web-pages and executing complex business workflows). Machine learning tasks are often compute intensive, making them similar to traditional high-performance computing (HPC) applications. Thus, large learning workloads perform very well on accelerated systems such as general purpose graphics processing units (GPU) or field-programmable gate arrays (FPGA) that have been used in the HPC field for more than



(a) Node Count

(b) Communication Layer

Fig. 5. Characteristics of Deep Learning Clusters

a decade now. Those devices focus on compute throughput by specializing their architecture to utilize the high data parallelism in HPC workloads. As we will see later, most learning researchers utilize accelerators such as GPUs or FPGAs for their computations. We emphasize that the main technique for acceleration is to exploit the inherent parallelism in learning workloads.

Out of the 227 reviewed papers, 147 papers present empirical results and provide details about their hardware setup. Fig. 4a shows a summary of the machine architectures used in research papers over the years. We see a clear trend towards GPUs, which dominate the publications beginning from 2013. However, even accelerated nodes are not sufficient for the large computational workload. Fig. 4b illustrated the quickly growing multi-node parallelism in those works. This shows that, beginning from 2015, distributed-memory architectures with accelerators such as GPUs have become the default option for machine learning at all scales today.

2.3.2 Multi-machine Parallelism. Training large-scale models is a very compute-intensive task. Thus, single machines are often not capable to finish this task in a desired time-frame. To accelerate the computation further, the computation can be distributed across multiple machines connected by a network. The most important metrics for the interconnection network (short: interconnect) are latency, bandwidth, and message-rate. Different network technologies provide different performance. For example, both modern Ethernet and InfiniBand provide high bandwidth but InfiniBand has significantly lower latencies and higher message rates. Special-purpose HPC interconnection networks can achieve higher performance in all three metrics. Yet, network communication remains generally slower than intra-machine communication.

Fig. 5a shows a breakdown of the number of nodes used in deep learning research over the years. It started very high with the large-scale DistBelief run, reduced slightly with the introduction of powerful accelerators and is on a quick rise again since 2015 with the advent of large-scale deep learning. Out of the 227 reviewed papers, 73 make use of distributed-memory systems and provide details about their hardware setup. We observe that large-scale setups, similar to HPC machines, are commonplace and essential in today's training.

2.4 Parallel Programming

Programming techniques to implement parallel learning algorithms on parallel computers depend on the target architecture. They range from simple threaded implementations to OpenMP on single machines. Accelerators are usually programmed with special languages such as NVIDIA's CUDA,

OpenCL, or in the case of FPGAs using hardware design languages. Yet, the details are often hidden behind library calls (e.g., cuDNN or MKL-DNN) that implement the time-consuming primitives.

On multiple machines with distributed memory, one can either use simple communication mechanisms such as TCP/IP or Remote Direct Memory Access (RDMA). On distributed memory machines, one can also use more convenient libraries such as the Message Passing Interface (MPI) or Apache Spark. MPI is a low level library focused on providing portable performance while Spark is a higher-level framework that focuses more on programmer productivity.

Fig. 5b shows a breakdown of the different communication mechanisms that were specified in 55 of the 73 papers using multi-node parallelism. It shows how the community quickly recognized that deep learning has very similar characteristics than large-scale HPC applications. Thus, beginning from 2016, the established MPI interface became the de-facto portable communication standard in distributed deep learning.

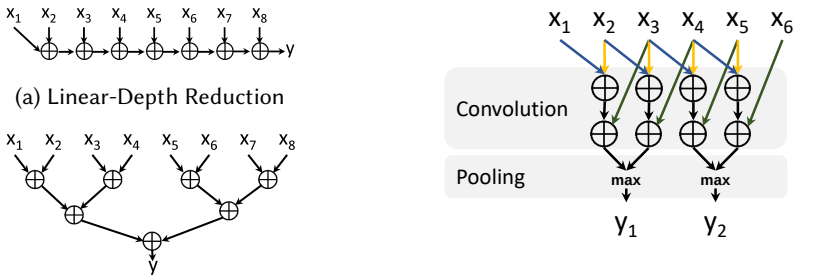
2.5 Parallel Algorithms

We now briefly discuss some key concepts in parallel computing that are needed to understand parallel machine learning. Every computation on a computer can be modeled as a directed acyclic graph (DAG). The vertices of the DAG are the computations and the edges are the data dependencies (or data flow). The computational parallelism in such a graph can be characterized by two main parameters: the graph's work $\mathbf{W}$, which corresponds to the total number of vertices and the graph's depth $\mathbf{D}$, which is the number of vertices on any longest path in the DAG. These two parameters allow us to characterize the computational complexity on a parallel system. For example, assuming we can process one operation per time unit, then the time needed to process the graph on a single processor is $T_1 = \mathbf{W}$ and the time needed to process the graph on an infinite number of processes is $T_\infty = \mathbf{D}$. The average parallelism in the computation is $\mathbf{W}/\mathbf{D}$, which is often a *good* number of processes to execute the graph with. Furthermore, we can show that the execution time of such a DAG on p processors is bounded by: $\min\{\mathbf{W}/p, \mathbf{D}\} \leq T_p \leq O(\mathbf{W}/p + \mathbf{D})$ [Arora et al. 1998; Brent 1974].

Most of the operations in learning can be modeled as operations on tensors (typically tensors as a parallel programming model [Solomonik and Hoefler 2015]). Such operations are highly data-parallel and only summations introduce dependencies. Thus, we will focus on parallel reduction operations in the following.

In a reduction, we apply a series of binary operators $\oplus$ to combine n values into a single value, e.g., $y = x_1 \oplus x_2 \oplus x_3 \cdots \oplus x_{n-1} \oplus x_n$. If the operation $\oplus$ is associative then we can change its application which changes the DAG from a linear-depth line-like graph as shown in Fig. 6a to a logarithmic-depth tree graph as shown in Fig. 6b. It is simple to show that the work and depth for reducing n numbers is $\mathbf{W} = n - 1$ and $\mathbf{D} = \log_2 n$, respectively. In deep learning, one often needs to reduce (sum) large tables of m independent parameters and return the result to all processes. This is called *allreduce* in the MPI specification [Gropp et al. 2014; Message Passing Interface Forum 2015].

In multi-machine environments, these tables are distributed across the machines which participate in the overall reduction operation. Due to the relatively low bandwidth between the machines (compared to local memory bandwidths), this operation is often most critical for distributed learning. We analyze the algorithms in a simplified LogP model [Culler et al. 1993], where we ignore injection rate limitations ($o = g = 0$), which makes it similar to the simple α-β model: $L = \alpha$ models the point-to-point latency in the network, $G = \beta$ models the cost per byte, and $P \leq p$ is the number of networked machines. Based on the DAG model from above, it is simple to show a lower bound for the reduction time $T_r \geq L \log_2(P)$ in this simplified model. Furthermore, because each element of the table has to be sent at least once, the second lower bound is $T_r \geq \gamma m G$, where γ represents the



(a) Linear-Depth Reduction

(b) Tree Reduction

(c) Convolution Downscaling in DNNs

Fig. 6. Reduction Schemes

size of a single data value and m is the number of values sent. This bound can be strengthened to $T_r \geq L\log_2(P) + 2\gamma mG(P-1)/P$ if we disallow redundant computations [Chan et al. 2007].

Several practical algorithms exist for the parallel allreduce operation in different environments and the best algorithm depends on the system, the number of processes, and the message size. We refer to Chan et al. [Chan et al. 2007] and Hoefler and Moor [Hoefler and Moor 2014] for a survey of key collective algorithms. Here, we summarize key algorithms that have been rediscovered in the context of parallel learning. The simplest algorithm is to combine two trees, one for summing the values to one process, similar to Fig. 6b, and one for broadcasting the values back to all processes; its complexity is $T_{tree} = 2\log_2(P)(L + \gamma mG)$. Yet, this algorithm is inefficient and can be optimized with a simple butterfly pattern, reducing the time to $T_{bfly} = \log_2(P)(L + \gamma mG)$. The butterfly algorithm is efficient (near-optimal) for small γm. For large γm and small P, a simple linear pipeline that splits the message into P segments is bandwidth-optimal and performs well in practice, even though it has a linear component in P: $T_{pipe} = 2(P-1)(L + \gamma \frac{m}{P} G)$. For most ranges of γm and P, one could use Rabenseifner's algorithm [Rabenseifner 2004], which combines reduce-scatter with gather, running in time $T_{rabe} = 2L\log_2(P) + 2\gamma mG(P-1)/P$. This algorithm achieves the lower bound but may be harder to implement and tune.

Other communication problems needed for convolutions and pooling, illustrated in Fig. 6c, exhibit high spatial locality due to strict neighbor interactions. They can be optimized using well-known HPC techniques for stencil computations such as MPI Neighborhood Collectives [Hoefler and Schneider 2012] (formerly known as sparse collectives [Hoefler and Traeff 2009]) or optimized Remote Memory Access programming [Belli and Hoefler 2015]. In general, exploring different low-level communication, message scheduling, and topology mapping [Hoefler and Snir 2011] strategies that are well-known in the HPC field could significantly speed up the communication in distributed deep learning.

2.6 Minibatch SGD

When performing SGD, it is common to decrease the number of weight updates by computing the sample loss in *minibatches*, averaging the gradient with respect to subsets of the data [Le et al. 2011]. Minibatches represent a trade-off between traditional SGD, which is proven to converge when drawing one sample at a time (as mentioned in Section 2.1), and batch methods [Nocedal and Wright 2006], which make use of the entire dataset at each iteration.

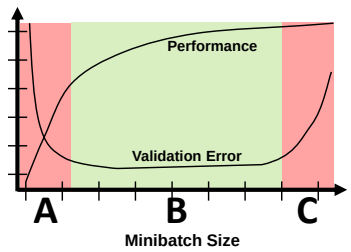
Minibatch SGD, shown in Algorithm 2, optimizes the training loss of a DNN with respect to groups of B samples from the dataset S. To compute the gradient of a DNN, we apply the widely-used backpropagation method [LeCun et al. 1989] on the sampled minibatch (lines 4-5).

Algorithm 2 Minibatch Stochastic Gradient Descent with Backpropagation

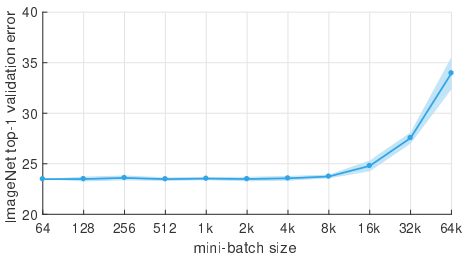
for $t = 0$ **to** $\frac{|S|}{B} \cdot epochs$ **do**
  $\vec{z} \leftarrow$ Sample B elements from S ▷ Obtain samples from dataset
  $w_{mb} \leftarrow w^{(t)}$ ▷ Load parameters
  $f \leftarrow \ell(w_{mb}, \vec{z}, h(\vec{z}))$ ▷ Compute forward evaluation
  $g_{mb} \leftarrow \nabla\ell(w_{mb}, f)$ ▷ Compute gradient using backpropagation
  $\Delta w \leftarrow u(g_{mb}, w^{(0,\ldots,t)}, t)$ ▷ Weight update rule
  $w^{(t+1)} \leftarrow w_{mb} + \Delta w$ ▷ Store parameters
end for

Setting the minibatch size is a complex optimization space on its own merit, as it affects both statistical accuracy and hardware efficiency. As illustrated in Fig. 7a, minibatches should not be too small (region A), so as to harness inherent concurrency in evaluation of the loss function; nor should they be too large (region C), as the quality of training decays once increased beyond a certain point. This behavior is empirically shown for larger minibatch sizes in Fig. 7b. Therefore, the typical size of a minibatch lies between the orders of 10 and 10,000 [Goyal et al. 2017; Hoffer et al. 2017; Smith et al. 2017; You et al. 2017b].

In the past, minibatch sizes in deep learning did not increase beyond 256 samples, both due to memory constraints and accuracy degradation. However, due to (a) networks with smaller memory footprint [Huang et al. 2017], (b) adaptive choice of learning rates [You et al. 2017b], and (c) the definition of a "warmup" phase [Goyal et al. 2017]; it became possible to train DNNs with minibatch sizes of 8k and 32k samples without significant loss in accuracy. Recent works even treat minibatch size as an adaptive hyper-parameter, increasing it as training progresses [Smith et al. 2017]. Overall, such works increase the upper bound on feasible minibatch sizes, but do not remove it.



(a) Minibatch Effect on Accuracy and Performance (Illustration)

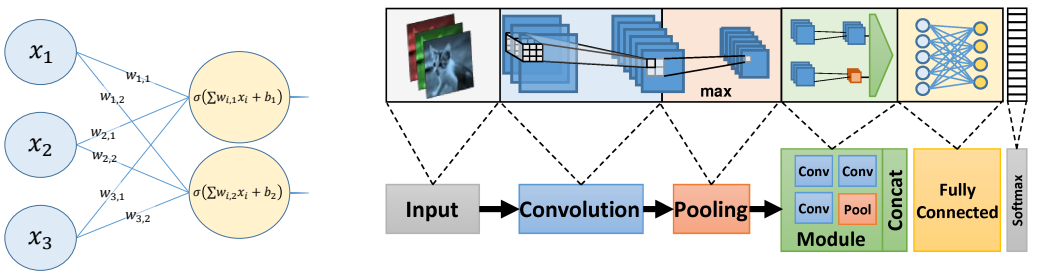


(b) Empirical Accuracy (ResNet-50, figure adapted from [Goyal et al. 2017], lower is better)

Fig. 7. Minibatch Size Effect on Accuracy and Performance

3 DEEP NEURAL NETWORKS

We now describe the anatomy of a Deep Neural Network (DNN). In Fig. 8, we see a DNN in two scales: the single layer (Fig. 8a) and the composition of such layers in a deep network (Fig. 8b). In the rest of this section, we describe popular layer types and their properties, followed by the computational description of deep networks and the backpropagation algorithm. Then, we study several examples of popular neural networks in detail, highlighting the computational trends driven by their definition.



(a) Neural Network Layer (b) Deep Network

Fig. 8. Deep Neural Network Architecture

3.1 Neurons

The basic building block of a deep neural network is the *neuron*. Modeled after the brain, an artificial neuron (Fig. 8a) accumulates signals from other neurons connected by *synapses*. An *activation function* (or *axon*) is applied on the accumulated value, which adds nonlinearity to the network and determines the signal this neuron "fires" to its neighbors. In *feed-forward neural networks*, the neurons are grouped to *layers* strictly connected to neurons in subsequent layers. In contrast, *recurrent neural networks* allows back-connections within the same layer.

3.1.1 Feed-Forward Layers. Neural network layers are implemented as weighted sums, using the synapses as weights. Activations (denoted σ) can be implemented as different functions, such as Sigmoid, Softmax, hyperbolic tangents, Rectified Linear Units (ReLU), or variants thereof [He et al. 2015]. When color images are used as input (as is commonly the case in computer vision), they are usually represented as a 4-dimensional tensor sized $N{\times}C{\times}H{\times}W$. As shown in Fig. 9, N is number of images in the minibatch, where each $H{\times}W$ image contains C channels (e.g., image RGB components). If a layer disregards spatial locality in the image tensor (e.g., a fully connected layer), the dimensions are flattened to $N \times (C \cdot H \cdot W)$. In typical DNN and CNN constructions, the number of channels, as well as the width and height of an image, change from layer to layer using the operators defined below. We denote the input and output channels of a layer by C_{in} and C_{out} respectively.

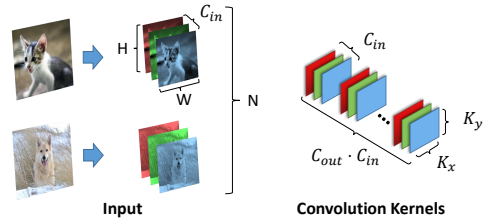
A fully connected layer (Fig. 8a) is defined on a group of neurons x (sized $N \times C_{in}$, disregarding spatial properties) by $y_{i,*} = \sigma(wx_{i,*} + b)$, where w is the weight matrix (sized $C_{in} \times C_{out}$) and b is a per-layer trainable bias vector (sized C_{out}). While this inner product is usually implemented with multiplication and addition, some works use other operators, such as similarity [Cohen et al. 2016a].

Not all layers in a neural network are fully connected. Sparsely connecting neurons and sharing weights is beneficial for reducing the number of parameters; as is the case in the popular *convolutional* layer. In a convolutional layer, every 3D tensor x (i.e., a slice of the 4D minibatch tensor representing one image) is convolved with C_{out} kernels of size $C_{in}{\times}K_y{\times}K_x$, where the base formula for a minibatch is given by:

$$y_{i,j,k,l} = \sum_{m=0}^{C_{in}-1} \sum_{k_y=0}^{K_y-1} \sum_{k_x=0}^{K_x-1} x_{i,j,m+k_y,l+k_x} \cdot w_{k,m,k_y,k_x}, \tag{2}$$

Name	Description
N	Minibatch size
C	Number of channels/neurons
H	Image Height
W	Image Width
K_x	Convolution kernel width
K_y	Convolution kernel height

(a) Data Dimensions



(b) Illustrative Example

Fig. 9. Summary of Layer Types and Data Dimensions

where y's dimensions are $N \times C_{out} \times H' \times W'$, $H' = H - K_y + 1$, and $W' = W - K_x + 1$, accounting for the size after the convolution, which does not consider cases where the kernel is out of the image bounds. Note that the formula omits various extensions of the operator [Dumoulin and Visin 2016], such as variable stride, padding, and dilation [Yu and Koltun 2016], each of which modifies the accessed indices and H', W'. The two inner loops of Eq. 2 are called the *convolution kernel*, and the kernel (or filter) size is $K_x \times K_y$.

While convolutional layers are the most computationally-intensive in DNNs, other layer types are prominently used in networks. Two such layers are the *pooling* and the *batch normalization* layers. The former reduces an input tensor in the width and height dimensions, performing an operation on contiguous sub-regions of the reduced dimensions, such as maximum (called max-pooling) or average, and is given by:

$$y_{i,j,k,l} = \max_{k_x \in [0,K_x), k_y \in [0,K_y)} x_{i+k_x, y+k_y, c, l}.$$

The goal of this layer is to reduce the size of a tensor by sub-sampling it while emphasizing important features. Applying subsequent convolutions of the same kernel size on a sub-sampled tensor enables learning high-level features that correspond to larger regions in the original data.

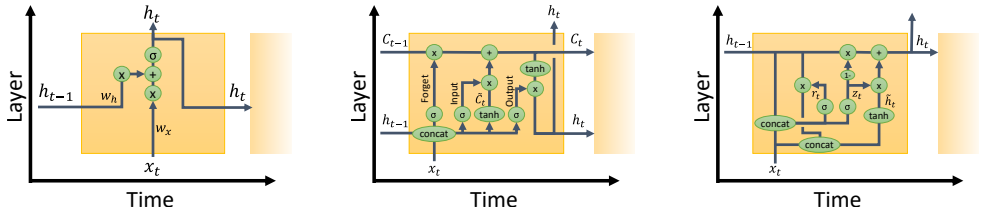
Batch Normalization (BN) [Ioffe and Szegedy 2015] is an example of a layer that creates inter-dependencies between samples in the same minibatch. Its role is to center the samples around a zero mean and a variance of one, which, according to the authors, reduces the internal covariate shift. BN is given by the following transformation:

$$y_{i,j,k,l} = \left(\frac{x_{i,j,k,l} - \mathbb{E}\left[x_{*,j,k,l}\right]}{\sqrt{\mathrm{Var}\left[x_{*,j,k,l}\right] + \epsilon}} \right) \cdot \gamma + \beta,$$

where γ, β are scaling factors, and ϵ is added to the denominator for numerical stability.

3.1.2 Recurrent Layers. Recurrent Neural Networks (RNNs) [Elman 1990] enable connections from a layer's output to its own inputs. These connections create "state" in the neurons, retaining persistent information in the network and allowing it to process data sequences instead of a single tensor. We denote the input tensor at time point t as $x^{(t)}$.

The standard Elman RNN layer is defined as $y^{(t)} = w_y \cdot \left(w_h \cdot h_{t-1} + w_x \cdot x^{(t)} \right)$ (omitting bias, illustrated in Fig. 10a), where h_t represents the "hidden" data at time-point t and is carried over to the next time-point. Despite the initial success of these layers, it was found that they tend to "forget" information quickly (as a function of sequence length) [Bengio et al. 1994]. To address this issue, Long-Short Term Memory (LSTM) [Hochreiter and Schmidhuber 1997] (Fig. 10b) units



(a) Recurrent Units (b) Long Short-Term Memory (c) Gated Recurrent Unit

Fig. 10. Recurrent Neural Network (RNN) Layers. Sub-figures (b) and (c) adapted from [Olah 2015].

redesign the structure of the recurrent connection to resemble memory cells. Several variants of LSTM exist, such as the Gated Recurrent Unit (GRU) [Cho et al. 2014] (Fig. 10c), which simplifies the LSTM gates to reduce the number of parameters.

3.2 Deep Networks

According to the definition of a fully connected layer, the expressiveness of a "shallow" neural network is limited to a separating hyperplane, skewed by the nonlinear activation function. When composing layers one after another, we create deep networks (as shown in Fig. 8b) that can approximate arbitrarily complex continuous functions. While the exact class of expressible function is currently an open problem, results [Cohen et al. 2016b; Delalleau and Bengio 2011] show that neural network depth can reduce breadth requirements exponentially with each additional layer.

A Deep Neural Network (DNN) can be represented as a function composition, e.g., $\ell(L_M(w_M, \cdots L_2(w_2, L_1(w_1, x))))$, where each function L_i is a layer, as described in Section 3.1, and each vector w_i represents layer i's weights (parameters). In addition to direct composition, a DNN DAG might reuse the output values of a layer in multiple subsequent layers, forming *shortcut* connections [He et al. 2016; Huang et al. 2017].

Computation of the DNN loss gradient $\nabla\ell$, which is necessary for SGD, can be performed by repeatedly applying the chain rule in a process commonly referred to as *backpropagation*. As shown in Fig. 11, the process of obtaining $\nabla\ell(w, x)$ is performed in two steps. First, $\ell(w, x)$ is computed by forward evaluation (left-hand side of the figure), computing each layer after its dependencies in a topological ordering. After computing the loss, information is propagated backward through the network (right-hand side of the figure), computing two gradients — ∇x (w.r.t. input data), and ∇w_i (w.r.t. layer weights). Note that some layers do not maintain mutable parameters (e.g., pooling, concatenation), and thus ∇w_i is not always computed.

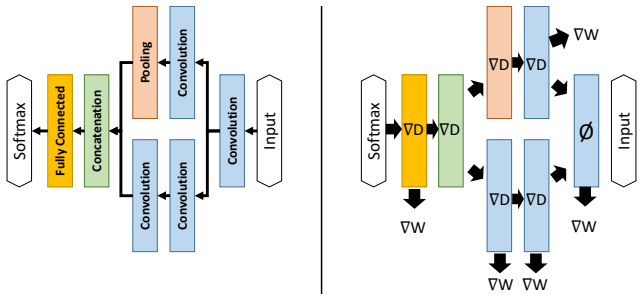


Fig. 11. The Backpropagation Algorithm

Table 4. Asymptotic Work-Depth Characteristics of DNN Layers

Layer Type	Eval.	Work (W)	Depth (D)
Activation	y	$O(NCHW)$	$O(1)$
	∇w	$O(NCHW)$	$O(1)$
	∇x	$O(NCHW)$	$O(1)$
Fully Connected	y	$O(C_{out} \cdot C_{in} \cdot N)$	$O(\log C_{in})$
	∇w	$O(C_{in} \cdot N \cdot C_{out})$	$O(\log N)$
	∇x	$O(C_{in} \cdot C_{out} \cdot N)$	$O(\log C_{out})$
Convolution (Direct)	y	$O(N \cdot C_{out} \cdot C_{in} \cdot H' \cdot W' \cdot K_x \cdot K_y)$	$O(\log K_x + \log K_y + \log C_{in})$
	∇w	$O(N \cdot C_{out} \cdot C_{in} \cdot H' \cdot W' \cdot K_x \cdot K_y)$	$O(\log K_x + \log K_y + \log C_{in})$
	∇x	$O(N \cdot C_{out} \cdot C_{in} \cdot H \cdot W \cdot K_x \cdot K_y)$	$O(\log K_x + \log K_y + \log C_{in})$
Pooling	y	$O(NCHW)$	$O(\log K_x + \log K_y)$
	∇w	–	–
	∇x	$O(NCHW)$	$O(1)$
Batch Normalization	y	$O(NCHW)$	$O(\log N)$
	∇w	$O(NCHW)$	$O(\log N)$
	∇x	$O(NCHW)$	$O(\log N)$

In terms of concurrency, we use the Work-Depth (W-D) model to formulate the costs of computing the forward evaluation and backpropagation of different layer types. Table 4 shows that the work (**W**) performed in each layer asymptotically dominates the maximal operation dependency path (**D**), which is at most logarithmic in the parameters. This result reaffirms the state of the practice, in which parallelism plays a major part in the feasibility of evaluating and training DNNs.

As opposed to feed-forward networks, RNNs contain self-connections and thus cannot be trained with backpropagation alone. The most popular way to solve this issue is by applying *backpropagation through time* (BPTT) [Werbos 1990], which unrolls the recurrent layer up to a certain amount of sequence length, using the same weights for each time-point. This creates a larger, feed-forward network that can be trained with the usual means.

In addition to the DNN loss function, it is common practice to *regularize* the objective function in order to promote generalization and avoid "exploding" gradient values. There are several ways to regularize the DNN training process. First, it is possible to apply Tikhonov regularization [Tikhonov 1995] (also referred to as *weight decay* or ridge regression), adding normalized parameter values as a separate term to the objective function, i.e., $\min f(x) + \|Cx\|^2$, where C is a normalization matrix and either the ℓ_1 or ℓ_2 norm is used. Second, *gradient clipping* [Pascanu et al. 2013] can be applied to normalize derivatives that are larger than a threshold, multiplying it by $\frac{thres}{\|\nabla \ell\|}$. This technique has proven to be useful when training RNNs [Pascanu et al. 2013] and handling asynchronous updates in distributed deep learning (see Section 6.3).

3.3 Case Studies

To understand how successful neural architectures orchestrate the aforementioned layers, we discuss five influential networks and highlight trends in their characteristics over the past years. Each of the networks, listed in Table 5, has achieved state-of-the-art performance upon publication. The table summarizes these networks, their computational characteristics, and their achieved test accuracy on the ImageNet [Deng et al. 2009] (1,000 class challenge) and CIFAR-10 [Krizhevsky 2009] datasets.

The listed networks, as well as other works [Chen et al. 2017; Coates et al. 2013; Dean et al. 2012; Han et al. 2016; Iandola et al. 2016b; Lin et al. 2014; Simonyan and Zisserman 2015; Zoph

Table 5. Popular Neural Network Characteristics

Property	LeNet	AlexNet	GoogLeNet	ResNet	DenseNet
Publication	[LeCun et al. 1998]	[Krizhevsky et al. 2012]	[Szegedy et al. 2015]	[He et al. 2016]	[Huang et al. 2017]
$\|w\|$	60K	61M	6.8M	1.7M–60.2M	~15.3M–30M
Layers ($\propto$ **D**)	7	13	27	50–152	40–250
Operations ($\propto$ **W**) (ImageNet-1k)	N/A	725M	1566M	~1000M–2300M	~600M–1130M
Top-5 Error (ImageNet-1k)	N/A	15.3%	9.15%	5.71%	5.29%
Top-1 Error (CIFAR-10)	N/A	N/A	N/A	6.41%	3.62%

et al. 2017], indicate three periods in the history of classification neural networks: experimentation (~1985–2010), growth (2010–2015), and resource conservation (2015–today).

In the experimentation period, different types of neural network structures (e.g., Deep Belief Networks [Bengio et al. 2007]) were researched, and the methods to optimize them (e.g., backpropagation) were developed. Once the neural network community has converged on deep feedforward networks (with the success of AlexNet cementing this decision), research during the growth period yielded networks with larger sizes and more operations in attempt to solve increasingly complex problems. This trend was supported by the advent of GPUs and other large computational resources (e.g., the Google Brain cluster).

However, as over-parameterization leads to overfitting, and since the resulting networks were too large to fit into consumer devices, efforts to decrease resource usage started around 2015. Research is since focused on increasing expressiveness, mostly by producing deeper networks, while also reducing the number of parameters and operations required to forward-evaluate the given network. By reducing memory and increasing energy efficiency, the resource conservation trend aims to move neural processing to the end user, i.e., to embedded and mobile devices. At the same time, smaller networks are faster to prototype and require less information to communicate when training on distributed platforms.

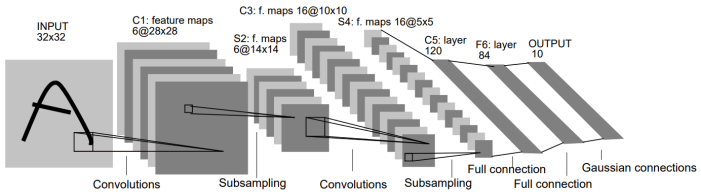


Fig. 12. The LeNet-5 Convolutional Neural Network (adapted from [LeCun et al. 1998])

3.3.1 LeNet [LeCun et al. 1998]. The first successful convolutional neural network, which was designed to identify hand-written digits in the MNIST dataset [LeCun et al. 1989; LeCun and Cortes 1998]. As shown in Fig. 12, LeNet-5 takes a single-channel 2D input, performs a series of 6 convolutions, then subsamples the filtered images by max-pooling. This convolution-pooling layer sequence occurs again, followed by 2 fully connected layers and a final fully connected softmax layer to produce the results.

For inference (forward evaluation) of an image, analyzing this network with respect to average parallelism ($\mathbf{W}/\mathbf{D}$) yields the following result (see Appendix A for details regarding each layer):

$$\begin{aligned}\mathbf{W} &= \mathbf{W}_{conv(32,5,1,6)} + \mathbf{W}_{pool(28,2,6)} + \mathbf{W}_{conv(14,5,6,16)} + \mathbf{W}_{pool(10,2,16)} + \\ &\quad \mathbf{W}_{fc(400,120)} + \mathbf{W}_{fc(120,84)} + \mathbf{W}_{fc(84,10)} \\ &= 117,600 + 14,112 + 470,400 + 4,800 + 48,000 + 10,080 + 840 \\ &= 665,832\end{aligned}$$

$$\mathbf{D} = 5 + 2 + 9 + 2 + 9 + 7 + 7 = 41$$

This indicates that even the simplest DNN exhibits high levels of concurrency, linearly increasing with the minibatch size.

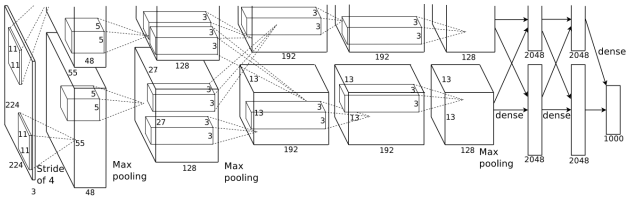


Fig. 13. AlexNet (adapted from [Krizhevsky et al. 2012])

3.3.2 AlexNet [Krizhevsky et al. 2012]. The AlexNet architecture was the winner of the ImageNet Large-Scale Visual Recognition (ILSVRC) 2012 [Deng et al. 2009] competition. Yielding nearly a twofold increase in accuracy over the preceding state-of-the-art (26.2% top-5 error), this network played a major role in the current state of DNN and Machine Learning.

Similar to LeNet, AlexNet contains a series of convolution-pooling layers followed by fully connected layers. However, it also uses sequences of convolutions and Local Response Normalization layers. Two major factors in the success of AlexNet are data augmentation and network regularization (Dropout) during training.

The network was implemented for GPUs (using a specialized cuda-convnet framework), training on ImageNet with a minibatch size of 128 images at a time.

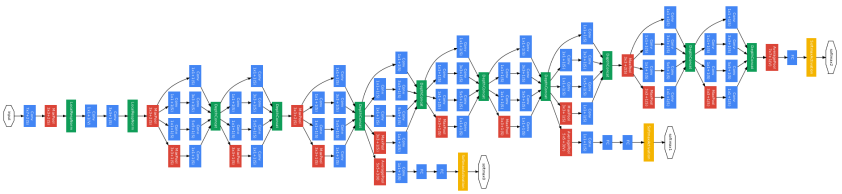


Fig. 14. The GoogLeNet (Inception-v1) Neural Architecture (adapted from [Szegedy et al. 2015])

3.3.3 GoogLeNet [Szegedy et al. 2015]. As opposed to the large number of parameters in AlexNet, the GoogLeNet architecture employs smaller series of convolutions organized in repeating *modules*. Inspired by the Network-in-Network [Lin et al. 2014] architecture, these modules invoke $1 \times 1 \times C_{in}$ convolutions (sometimes referred to as 1×1 convolutions). Such modules increase the expressiveness by increasing the depth of the DNN, and at the same time act as dimensionality reduction modules, essentially trading breadth (number of neurons per layer) for depth.

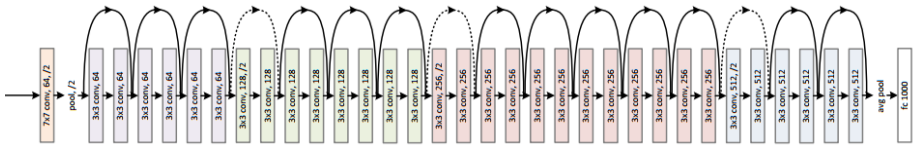
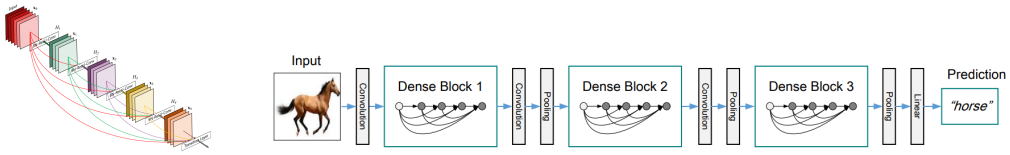


Fig. 15. ResNet (adapted from [He et al. 2016])

3.3.4 ResNet [He et al. 2016]. The trend of DNNs becoming deeper and narrower continued with successful networks like VGG [Simonyan and Zisserman 2015] (published at the same time as GoogLeNet), comprising up to 30 layers of convolutions. However, as networks increased in depth, their successful training had become harder.

The authors of ResNet address the depth issue by training a slightly different inter-layer interaction: instead of composing layers as described in Section 3.2, every convolutional module would add its input to the output (as shown in Fig. 15). Residuals are implemented as "shortcut" identity connections to the network. The system then trains layers with respect to their residuals instead of their original values, and, according to the authors, this solves the inherent degradation in accuracy as networks become deeper.

With ResNet, it became possible to train networks with depths of 50 to 152 layers, further increasing the quality of the results by allowing higher-level features to be learned.



(a) Dense Block (b) Full Network

Fig. 16. DenseNet (adapted from [Huang et al. 2017])

3.3.5 DenseNet [Huang et al. 2017]. Following the success of ResNets, DenseNets further increase the number of connections between layers. As opposed to the module identity shortcut, densely-connected blocks **concatenate** each layer's outputs to the inputs of the next layers, similar to [Deng et al. 2012]. According to the authors, this type of connection induces better gradient propagation, as features in subsequent layers are not required to be strictly high-level. Rather, since each layer also receives the inputs of the previous level, features can be constructed from both low-level information and the resulting high-level outputs.

The practical result is that with half the parameters and required operations, DenseNets achieve similar results as ResNets. When their depth is increased to 250 layers, DenseNets reduce the validation error by a factor of ~2 in CIFAR-10 compared to ResNets. This comes at the cost of increasing the number of parameters by almost an order of magnitude. Specifically, ResNets achieve 6.41% error with 1.7M parameters, whereas DenseNets achieve 3.62% error with 15.3M parameters.

3.3.6 Outlook. In summary, recent DNN architectures heavily base on convolutional operations, are very deep, and become narrower with time. While it is not known which direction the next breakthrough will take; efforts on DNN compression [Han et al. 2016] and more recent architectures [Chen et al. 2017; Real et al. 2018] seem to focus on decreasing the number of parameters and

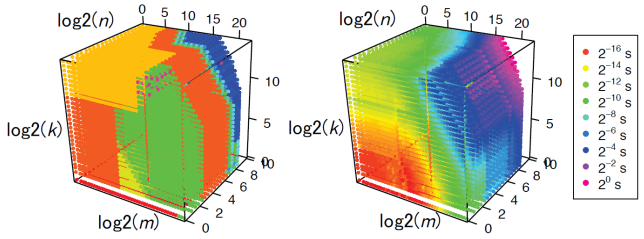


Fig. 17. Performance of `cublasSgemm` on an NVIDIA Tesla K80 for various matrix sizes (adapted from [Oyama et al. 2016])

operations, while maintaining the same accuracy or increasing it. Nevertheless, it is still necessary to use vast computational resources to train these networks, let alone find new networks automatically.

4 LAYER COMPUTATION

Given that neural network layers operate on 4-dimensional tensors (Fig. 9a) and the high locality of the operations, there are several opportunities for parallelizing layer execution. In most cases, computations (e.g., in the case of pooling layers) can be directly parallelized. However, in order to expose parallelism in other layer types, computations have to be reshaped. Below, we list efforts to model DNN performance, followed by a concurrency analysis of three popular DNN layer types.

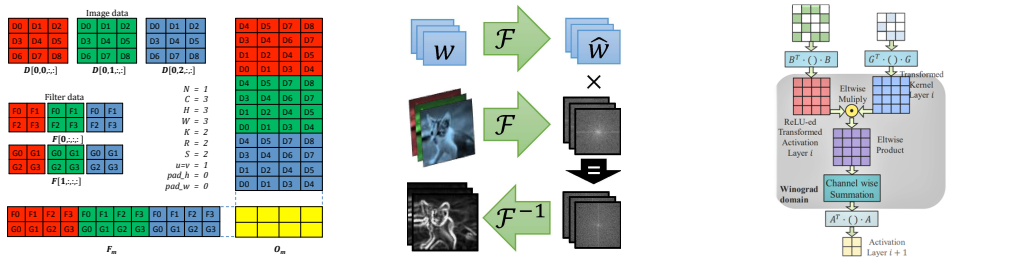
4.1 Performance Modeling

Even with work and depth models, it is hard to estimate the runtime of a single DNN layer, let alone an entire network. Fig. 17 presents measurements of the performance of `cublasSgemm`, a highly-tuned matrix multiplication implementation in the NVIDIA CUBLAS library [NVIDIA 2017a] that is in the core of all the layer types described below. The figure shows that as matrix dimensions are modified, the performance does not change linearly, and that in practice the system internally chooses from one of 15 implementations for the operation, where the left-hand side of the figure depicts the segmentation. As we shall show, there is a wide variety of approaches to implement convolutional layers, wherein the same principle applies.

In spite of the above observation, other works still manage to approximate the runtime of a given DNN with performance modeling. Using the values in the figure as a lookup table, it was possible to predict the time to compute and backpropagate through minibatches of various sizes with ~5–19% error, even on clusters of GPUs with asynchronous communication [Oyama et al. 2016]. The same was achieved for CPUs in a distributed environment [Yan et al. 2015], using a similar approach, and for Intel Xeon Phi accelerators [Viebke et al. 2017] strictly for training time estimation (i.e., not individual layers or DNN evaluation). Paleo [Qi et al. 2017] derives a performance model from operation counts alone (with 10–30% prediction error), and Pervasive CNNs [Song et al. 2017] uses performance modeling to select networks with decreased accuracy to match real-time requirements from users.

4.2 Fully Connected Layers

As described in Section 3.1, a fully connected layer can be expressed and modeled (see Table 4) as a matrix-matrix multiplication of the weights and the neuron values (x vector per sample in a minibatch). To that end, efficient linear algebra libraries, such as CUBLAS [NVIDIA 2017a] and MKL [Intel 2009], can be used. The BLAS [Netlib 2017] GEneral Matrix-Matrix multiplication



(a) `im2col` (adapted from [Chetlur et al. 2014])

(b) FFT

(c) Winograd (adapted from [Liu et al. 2018a])

Fig. 18. Convolutional Layer Methods of Computation

(GEMM) operator, used for this purpose, also includes scalar factors that enable matrix scaling and accumulation, which can be used when batching groups of neurons.

[Vanhoucke et al. 2011] present a variety of methods to further optimize CPU implementations of fully connected layers. In particular, the paper shows efficient loop construction, vectorization, blocking, unrolling, and batching. The paper also demonstrates how weights can be quantized to use fixed-point math instead of floating point.

4.3 Convolution

Convolutions constitute the majority of computations involved in training and inference of DNNs. As such, the research community and the industry have invested considerable efforts into optimizing their computation on all platforms. Fig. 18 depicts the convolution methods detailed below, and Table 6 summarizes their work and depth characteristics (see Appendix B for detailed analyses).

While a convolution layer (Eq. 2) can be computed directly, as the equation states, it will not fully utilize the resources of vector processors (e.g., Intel's AVX registers) and many-core architectures (e.g., GPUs), which are geared towards many parallel multiplication-accumulation operations. It is, however, possible to increase the utilization by introducing data redundancy or basis transformation.

The first algorithmic change proposed for convolutional layers was the use of the well-known technique to transform a discrete convolution into matrix multiplication, using *Toeplitz matrices* (colloquially known as *im2col*). The first occurrence of unrolling convolutions in CNNs can be found in [Chellapilla et al. 2006], where CPUs and GPUs were used for training (since the work precedes CUDA, it uses Pixel Shaders for GPU computations). This method was popularized by [Coates et al. 2013], and consists of reshaping the images in the minibatch from 3D tensors to 2D matrices. Each 1D row in the matrix contains an unrolled 2D patch that would usually be convolved (possibly with overlap), generating redundant information (see Fig. 18a). The convolution kernels are then stored as a 2D matrix, where each column represents an unrolled kernel (one convolution filter). Multiplying those two matrices results in a matrix that contains the convolved tensor in 2D format, which can be reshaped to 3D for subsequent operations. Note that this operation can be generalized to 4D tensors (an entire minibatch), converting it into a single matrix multiplication.

While processor-friendly, the GEMM method (as described above) consumes a considerable amount of memory, and thus was not scalable. Practical implementations of the GEMM method, such as in CUDNN [Chetlur et al. 2014], implement "implicit GEMM", in which the Toeplitz matrix is never materialized. It was also reported [Cong and Xiao 2014] that the Strassen matrix multiplication [Strassen 1969] can be used for the underlying computation, reducing the number of operations by up to 47%.

Table 6. Work-Depth Analysis of Convolution Implementations

Method	Work (W)	Depth (D)
Direct	$N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x$	$\lceil \log_2 C_{in} \rceil + \lceil \log_2 K_y \rceil + \lceil \log_2 K_x \rceil$
im2col	$N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x$	$\lceil \log_2 C_{in} \rceil + \lceil \log_2 K_y \rceil + \lceil \log_2 K_x \rceil$
FFT	$c \cdot HW \log_2(HW) \cdot (C_{out} \cdot C_{in} + N \cdot C_{in} + N \cdot C_{out}) + HWN \cdot C_{in} \cdot C_{out}$	$2\lceil \log_2 HW \rceil + \lceil \log_2 C_{in} \rceil$
Winograd ($m \times m$ tiles, $r \times r$ kernels)	$\alpha(r^2 + \alpha r + 2\alpha^2 + \alpha m + m^2) + C_{out} \cdot C_{in} \cdot P$ $(\alpha \equiv m - r + 1, \quad P \equiv N \cdot \lceil H/m \rceil \cdot \lceil W/m \rceil)$	$2\lceil \log_2 r \rceil + 4\lceil \log_2 \alpha \rceil + \lceil \log_2 C_{in} \rceil$

A second method to compute convolutions is to make use of the Fourier domain, in which convolution is defined as an element-wise multiplication [Mathieu et al. 2014; Vasilache et al. 2015]. In this method, both the data and the kernels are transformed using FFT, multiplied, and the inverse FFT is applied on the data, as in the formula below:

$$y_{i,j,*,*} = \mathcal{F}^{-1}\left(\sum_{k=0}^{C_{in}} \mathcal{F}\left(x_{i,k,*,*}\right) \circ \mathcal{F}\left(w_{j,k,*,*}\right)\right)$$

where $\mathcal{F}$ denotes the Fourier Transform and $\circ$ is element-wise multiplication. Note that for a single minibatch, it is enough to transform w once and reuse the results.

Experimental results [Vasilache et al. 2015] have shown that the larger the convolution kernels are, the more beneficial FFT becomes, yielding up to 16× performance over the GEMM method, which has to process patches of proportional size to the kernels. Additional optimizations were made to the FFT and IFFT operations in [Vasilache et al. 2015], using DNN-specific knowledge: (a) The process uses decimation-in-frequency for FFT and decimation-in-time for IFFT in order to mitigate bit-reversal instructions; (b) multiple FFTs with sizes ≤32 are batched together and performed at the warp-level on the GPU; and (c) pre-computation of twiddle factors.

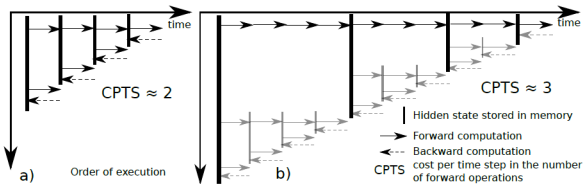
Working with DNNs, FFT-based convolution can be optimized further. In [Zlateski et al. 2016], the authors observed that due to zero-padding, the convolutional kernels, which are considerably smaller than the images, mostly consist of zeros. Thus, pruned FFT [Sorensen and Burrus 1993] can be executed for transforming the kernels, reducing the number of operations by 3×. In turn, the paper reports 5× and 10× speedups for CPUs and GPUs, respectively.

The prevalent method used today to perform DNN convolutions is Winograd's algorithm for minimal filtering [Winograd 1980]. First proposed by [Lavin and Gray 2016], the method modifies the original algorithm for multiple filters (as is the case in convolutional layers), performing the following computation for one tile:

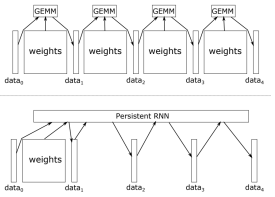
$$y_{i,j,*,*} = A^T\left(\sum_{m=0}^{C_{in}} G w_{j,m,*,*} G^T \circ B^T x_{i,m,*,*} B\right) A,$$

with the matrices A, G, B constructed as in Winograd's algorithm (full implementation in Appendix B).

Since the number of operations in Winograd convolutions grows quadratically with filter size, the convolution is decomposed into a sum of tiled, small convolutions, and the method is strictly used for small kernels (e.g., 3×3). Additionally, because the magnitude of elements in the expression



(a) Dynamic Programming BPTT (adapted from [Gruslys et al. 2016])



(b) Persistent RNNs (adapted from slides of [Diamos et al. 2016])

Fig. 19. RNN Optimizations

increases with filter size, the numerical accuracy of Winograd convolution is generally lower than the other methods, and decreases as larger filters are used.

Table 6 lists the concurrency characteristics of the aforementioned convolution implementations, using the Work-Depth model. From the table, we can see that each method exhibits different behavior, where the average parallelism ($\mathbf{W/D}$) can be determined by the kernel size or by image size (e.g., FFT). This coincides with experimental results [Chetlur et al. 2014; Lavin and Gray 2016; Vasilache et al. 2015], which show that there is no "one-size-fits-all" convolution method. We can also see that the Work and Depth metrics are not always sufficient to reason about absolute performance, as the Direct and im2col methods exhibit the same concurrency characteristics, even though `im2col` is faster in many cases, due to high processor utilization and memory reuse (e.g., caching) opportunities.

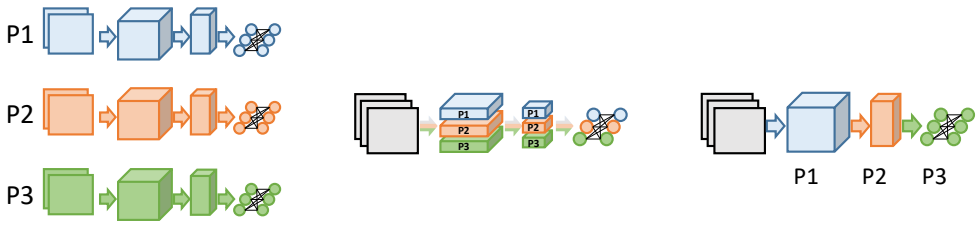
Data layout also plays a role in convolution performance. [Li et al. 2016] assert that convolution and pooling layers can be computed faster by transposing the data from $N \times C \times H \times W$ tensors to $C \times H \times W \times N$. The paper reports up to 27.9× performance increase over the state-of-the-art for a single layer, and 5.6× for a full DNN (AlexNet). The paper reports speedup even in the case of transposing the data during the computation of the DNN, upon inputting the tensor to the layer.

DNN primitive libraries, such as CUDNN [Chetlur et al. 2014] and MKLDNN [Intel 2017], provide a variety of convolution methods and data layouts. In order to assist users in a choice of algorithm, such libraries provide functions that choose the best-performing algorithm given tensor sizes and memory constraints. Internally, the libraries may run all methods and pick the fastest one.

4.4 Recurrent Units

The complex gate systems that occur within RNN units (e.g., LSTMs, see Fig. 10b) contain multiple operations, each of which incurs a small matrix multiplication or an element-wise operation. Due to this reason, these layers were traditionally implemented as a series of high-level operations, such as GEMMs. However, the acceleration of such layers by parallelism is possible. Moreover, since RNN units are usually chained together (forming consecutive recurrent layers), two types of concurrency can be considered: within the same layer, and between consecutive layers.

Appleyard et al. [Appleyard et al. 2016] describe several optimizations that can be implemented for GPUs. The basic optimization that can be performed is fusing all computations (GEMMs and otherwise) into one function (kernel), saving intermediate results in scratch-pad memory. This both reduces the kernel scheduling overhead, and conserves round-trips to the global memory, which is orders of magnitude slower than the scratch-pad memory. Other optimizations include pre-transposition of matrices and enabling concurrent execution of independent recurrent units on different multi-processors on the GPU. As for inter-layer optimizations, the paper implements pipelining of stacked RNN unit computations, immediately starting to propagate through the



(a) Data Parallelism (b) Model Parallelism (c) Layer Pipelining

Fig. 20. Neural Network Parallelism Schemes

next layer once its data dependencies have been met. Overall, these optimizations result in ~11× performance increase over the high-level implementation.

From the memory consumption perspective, dynamic programming was proposed [Gruslys et al. 2016] for RNNs (see Fig. 19a) in order to balance between caching intermediate results and recomputing forward inference for backpropagation. For long sequences (1000 time-points), the algorithm conserves 95% memory over standard BPTT, while adding ~33% time per iteration.

Persistent RNNs [Diamos et al. 2016] are an optimization that addresses two limitations of GPU utilization: small minibatch sizes and long sequences of inputs. By caching the weights of standard RNN units on the GPU registers, they optimize memory round-trips between timesteps (x_t) during training (Fig. 19b). In order for the registers not to be scheduled out, this requires the GPU kernels that execute the RNN layers to be "persistent", performing global synchronization on their own and circumventing the normal GPU programming model. The approach attains up to ~30× speedup over previous state-of-the-art for low minibatch sizes, performing on the order of multiple TFLOP/s per-GPU, even though it does not execute GEMM operations and loads more memory for each multi-processor. Additionally, the approach reduces the total memory footprint of RNNs, allowing users to stack more layers using the same resources.

5 SHARED-MEMORY PARALLELISM

The high average parallelism ($\mathbf{W}/\mathbf{D}$) in neural networks can not only be harnessed to compute individual layers efficiently, but also to evaluate the whole network concurrently with respect to different dimensions. Owing to the use of minibatches, the breadth ($\propto \mathbf{W}$) of the layers, and the depth of the DNN ($\propto \mathbf{D}$), it is possible to partition both the forward evaluation and the backpropagation phases (lines 4–5 in Algorithm 2) among parallel processors. Below, we discuss three prominent partitioning strategies, illustrated in Fig. 20 and summarized in Table 7: partitioning by input samples (data parallelism), by network structure (model parallelism), and by layer (pipelining).

5.1 Data Parallelism

In training, the data is computed in minibatches of size N. As most of the layers are independent with respect to N, a straightforward approach for parallelization is to partition the work of the minibatch samples to different processors. This method, called data parallelism (initially defined as pattern parallelism, as input samples were called patterns), dates back to the first practical implementations of artificial neural networks [Zhang et al. 1990].

5.1.1 Minibatch Size. The use of minibatches in SGD for DNNs may have initially been driven by data parallelism. In [Farber and Asanovic 1997], multiple vector accelerator microprocessors (Spert-II) were used for parallelizing error backpropagation for neural network training. To support

Table 7. Overview of Shared-Memory Parallelism in Deep Learning

Category	Papers
Data Parallelism	[Farber and Asanovic 1997; Goyal et al. 2017; Jiang et al. 2017; Krizhevsky et al. 2012; Le et al. 2011; Raina et al. 2009; Smith et al. 2017; Yadan et al. 2013; You et al. 2017b; Zhang and Chen 2014; Zhang et al. 1990; Zinkevich et al. 2010; Zlateski et al. 2016]
Model Parallelism	[Coates et al. 2013; Ericson and Mbuvha 2017; Lee et al. 2015; Muller and Gunzinger 1994; Ngiam et al. 2010; Yadan et al. 2013]
Layer Pipelining	[Chilimbi et al. 2014; Dean et al. 2012; Deng et al. 2012]
Hybrid Parallelism	[Ben-Nun et al. 2015; Chilimbi et al. 2014; Dean et al. 2012; Gaunt et al. 2017; Krizhevsky 2014; Viebke et al. 2017]

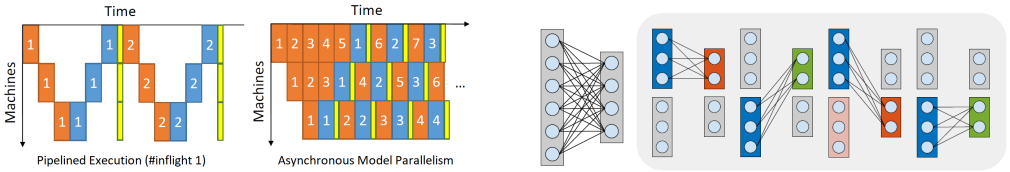
data parallelism, the paper presents a version of delayed gradient updates that they call "bunch mode", where the gradient is updated several times prior to updating the weights. This method is essentially equivalent to minibatch SGD.

Data parallelism partitions a given minibatch, or set of images for inference, among multiple computational resources (cores or devices). Apart from Batch Normalization (BN) [Ioffe and Szegedy 2015], all layers operate on a single sample at a time, so forward evaluation and backpropagation are almost completely independent. In the weight update phase, however, the results of the partitions have to be averaged to obtain the gradient w.r.t. the minibatch, which potentially induces an allreduce operation. Furthermore, in this partitioning method, the memory of all the DNN parameters has to be accessible for all participating devices, which means that, e.g., it may need to be duplicated among GPUs.

One of the earliest occurrences of mapping data parallel DNN computations to GPUs were performed in [Raina et al. 2009]. The paper focuses on the problem of training Deep Belief Networks [Hinton et al. 2006], mapping the unsupervised training procedure to GPUs by running minibatch SGD. The paper shows speedup of up to 72.6× over CPU when training Restricted Boltzmann Machines. Today, data parallelism is supported by the vast majority of deep learning frameworks, using a single GPU, multiple GPUs, or a cluster of multi-GPU nodes.

The scaling of data parallelism is naturally limited by the minibatch size. By applying various modifications (Section 2.6), recent works have successfully managed to increase the batch size to 8k samples [Goyal et al. 2017], 32k samples [You et al. 2017b], and even 64k [Smith et al. 2017] without losing accuracy. Thus, the issue is not as severe as claimed in prior works [Seide et al. 2014b]. One bottleneck that hinders scaling of data parallelism, however, is the BN layer, which requires a full synchronization point upon invocation. Since BN layers recur multiple times in some DNN architectures [He et al. 2016], this is too costly. Thus, popular implementations of BN follow the approach driven by [Goyal et al. 2017; Hoffer et al. 2017; You et al. 2017b], in which small subsets (e.g., 32 samples) of the minibatch are normalized independently. If at least 32 samples are scheduled to each processor, then this synchronization point is local, which in turn increases scaling.

Another approach to the BN problem is to define a different layer altogether. Weight Normalization (WN) [Salimans and Kingma 2016] proposes to separate the parameter (w) norm from its directionality by way of re-parameterization. In WN, the weights are defined as $w = \left(\frac{g}{\|v\|}\right) \cdot v$, where g represents weight magnitude and v a normalized direction (as changing the magnitude of v will not introduce changes in $\nabla\ell$). This layer essentially reduces the depth ($\mathbf{D}$) of the operator from $O(\log N)$ to $O(1)$, removing inter-dependencies within the minibatch. According to the authors,



(a) Pipelined synchronous execution vs. asynchronous (adapted from [Gaunt et al. 2017]).

(b) Decomposing a convolutional layer into sub-layers (adapted from [Zlateski et al. 2016]).

Fig. 21. Data Parallelism Schemes

WN reduces the need for BN, achieving comparable accuracy using a simplified version of BN (without variance correction).

5.1.2 Other Approaches. Additional approaches for data parallelism were proposed in literature. In ParallelSGD [Zinkevich et al. 2010], SGD is run (possibly with minibatches) k times in parallel, dividing the dataset among the processors. After the convergence of all SGD instances, the resulting weights are aggregated and averaged to obtain $\overline{w}$, resembling ensemble learning.

ParallelSGD [Zinkevich et al. 2010], as well as other deep learning implementations [Jiang et al. 2017; Le et al. 2011; Zhang and Chen 2014], were designed with the MapReduce [Dean and Ghemawat 2008] programming paradigm. Using MapReduce, it is easy to map parallel tasks onto multiple processors, as well as distributed environments, e.g., clusters. Prior to these works, the scaling and potential application of MapReduce were studied [Chu et al. 2007] on a variety of machine learning problems, including NNs, promoting the need to shift from single-processor learning to distributed memory systems.

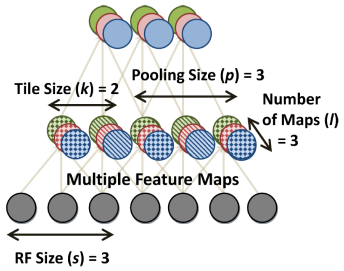
While the MapReduce model was successful for deep learning at first, its generality hindered DNN-specific optimizations. Therefore, current implementations make use of high-performance communication interfaces (e.g., MPI) to implement features such as asynchronous execution and pipelining [Gaunt et al. 2017] (Fig. 21a), sparse communication (see Section 6.3), and fine-grained parallelism between computational resources [Zlateski et al. 2016]. In the latter, minibatches are fragmented into sub-batches (Fig. 21b) for hybrid CPU-GPU inference, as well as memory footprint reduction.

5.2 Model Parallelism

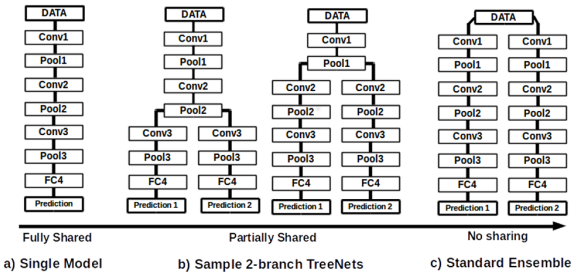
The second partitioning strategy for DNN training is model parallelism (also known as network parallelism). This strategy divides the work according to the neurons in each layer, namely the C, H, or W dimensions in a 4-dimensional tensor. In this case, the sample minibatch is copied to all processors, and different parts of the DNN are computed on different processors, which can conserve memory (since the full network is not stored in one place) but incurs additional communication after every layer.

A major differentiating point of model parallelism from data parallelism is that there is no trade-off between accuracy and scaling, as the minibatch size does not change. Nevertheless, the DNN architecture creates layer interdependencies, which, in turn, generate communication that determines the overall performance. Fully connected layers, for instance, incur all-to-all communication (as opposed to allreduce in data parallelism), as neurons connect to all the neurons of the next layer.

To reduce communication costs in fully connected layers, it has been proposed [Muller and Gunzinger 1994] to introduce redundant computations to neural networks. In particular, this method



(a) Locally Connected Network (adapted from [Ngiam et al. 2010]).



(b) TreeNets (adapted from [Lee et al. 2015]).

Fig. 22. Model Parallelism Schemes

partitions an NN such that each processor will be responsible for twice the neurons (with overlap), and thus would need to compute more but communicate less.

Another method proposed for reducing communication in fully connected layers is to use Cannon's matrix multiplication algorithm, modified for DNNs [Ericson and Mbuvha 2017]. The paper reports that Cannon's algorithm produces better efficiency and speedups over simple partitioning on small-scale multi-layer fully connected networks.

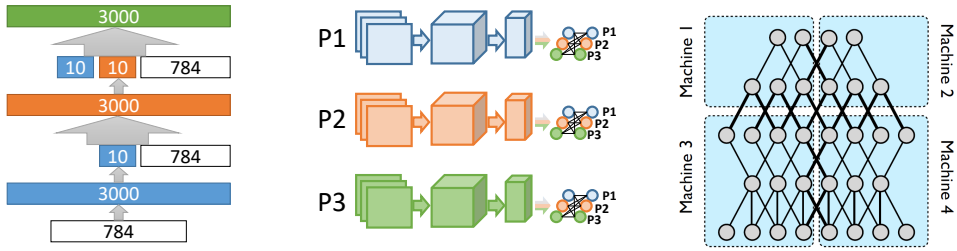
As for CNNs, using model parallelism for convolutional layers is relatively inefficient. If samples are partitioned across processors, then each convolutional layer would have to obtain all results from the other processors to compute the convolution, as the operation sums over all channels. To mitigate this problem, Locally Connected Networks (LCNs) [Ngiam et al. 2010] were introduced. While still performing convolutions, LCNs define multiple local filters for each region (Fig. 22a), enabling partitioning by the C, H, W dimensions that does not incur all-to-all communication.

Using LCNs and model parallelism, the work presented in [Coates et al. 2013] managed to outperform a CNN of the same size running on 5,000 CPU nodes with a 3-node multi-GPU cluster. Due to LCN not behaving as CNNs, i.e., there is no weight sharing apart from spatial image boundaries, training is not communication-bound, and scaling can be achieved. Successfully applying the same techniques on CNNs requires fine-grained control over parallelism, as we shall show in Section 5.4. Unfortunately, weight sharing is an important part of CNNs, contributing to memory footprint reduction as well as improving generalization, and thus standard convolutional layers are seen more frequently than LCNs.

A second form of model parallelism is the replication of DNN elements. In TreeNets [Lee et al. 2015], the authors study ensembles of DNNs (groups of separately trained networks whose results are averaged, rather than their parameters), and propose a mid-point between ensembles and training a single model: a certain layer creates a "junction", from which multiple copies of the network are trained (see Fig. 22b). The paper defines ensemble-aware loss functions and backpropagation techniques, so as to regularize the training process. The training process, in turn, is parallelized across the network copies, assigning each copy to a different processor. The results presented in the paper for three datasets indicate that TreeNets essentially train an ensemble of expert DNNs.

5.3 Pipelining

In deep learning, pipelining can either refer to overlapping computations, i.e., between one layer and the next (as data becomes ready); or to partitioning the DNN according to depth, assigning



(a) Deep Stacking Network (adapted from [Deng et al. 2012])

(b) Hybrid Data/Model Parallelism

(c) DistBelief Model Replica (adapted from [Dean et al. 2012])

Fig. 23. Pipelining and Hybrid Parallelism Schemes

layers to specific processors (Fig. 20c). Pipelining can be viewed as a form of data parallelism, since elements (samples) are processed through the network in parallel, but also as model parallelism, since the length of the pipeline is determined by the DNN structure.

The first form of pipelining can be used to overlap forward evaluation, backpropagation, and weight updates. This scheme is widely used in practice [Abadi et al. 2015; Collobert et al. 2011; Jia et al. 2014], and increases utilization by mitigating processor idle time. In a finer granularity, neural network architectures can be designed around the principle of overlapping layer computations, as is the case with Deep Stacking Networks (DSN) [Deng et al. 2012]. In DSNs, each step computes a different fully connected layer of the data. However, the results of all previous steps are concatenated to the layer inputs (see Fig. 23a). This enables each layer to be partially computed in parallel, due to the relaxed data dependencies.

As for layer partitioning, there are several advantages for a multi-processor pipeline over both data and model parallelism: (a) there is no need to store all parameters on all processors during forward evaluation and backpropagation (as with model parallelism); (b) there is a fixed number of communication points between processors (at layer boundaries), and the source and destination processors are always known. Moreover, since the processors always compute the same layers, the weights can remain cached to decrease memory round-trips. Two disadvantages of pipelining, however, are that data (samples) have to arrive at a specific rate in order to fully utilize the system, and that latency proportional to the number of processors is incurred.

In the following section, we discuss two well-known implementations of layer partitioning — DistBelief [Dean et al. 2012] and Project Adam [Chilimbi et al. 2014] — which combine the advantages of pipelining with data and model parallelism.

5.4 Hybrid Parallelism

The combination of multiple parallelism schemes can overcome the drawbacks of each scheme. Below we overview successful instances of such hybrids.

In AlexNet, most of the computations are performed in the convolutional layers, but most of the parameters belong to the fully connected layers. When mapping AlexNet to a multi-GPU node using data or model parallelism separately, the best reported speedup for 4 GPUs over one is ~2.2× [Yadan et al. 2013]. One successful example [Krizhevsky 2014] of a hybrid scheme applies data parallelism to the convolutional layer, and model parallelism to the fully connected part (see Fig. 23b). Using this hybrid approach, a speedup of up to 6.25× can be achieved for 8 GPUs over one,

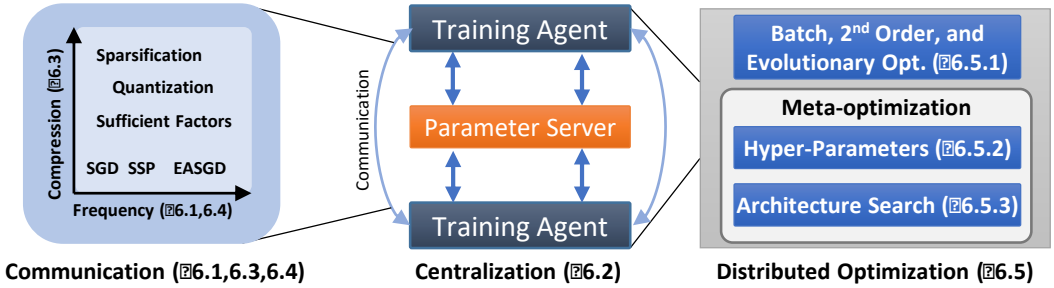


Fig. 24. Distributed Deep Learning Section Overview

with less than 1% accuracy loss. These results were also reaffirmed in other hybrid implementations [Ben-Nun et al. 2015], in which 3.1× speedup was achieved for 4 GPUs using the same approach.

AMPNet [Gaunt et al. 2017] is an asynchronous implementation of DNN training on CPUs. By defining an intermediate representation for neural networks that includes control-flow constructs, model parallelism can be achieved. In particular, internal fine-grained parallel tasks within and between layers are identified and scheduled asynchronously. Additionally, asynchronous execution of dynamic control flow enables pipelining the tasks of forward evaluation, backpropagation, and weight update (Fig. 21a, right). The main advantage of AMPNet is in recurrent, tree-based, and gated-graph neural networks, all of which exhibit heterogeneous characteristics, i.e., variable length for each sample and dynamic control flow (as opposed to homogeneous CNNs). The paper shows speedups of up to 3.94× over the TensorFlow [Abadi et al. 2015] framework.

Lastly, the DistBelief [Dean et al. 2012] distributed deep learning system combines all three parallelism strategies. In the implementation, training is performed on multiple model replicas simultaneously, where each replica is trained on different samples (data parallelism). Within each replica (shown in Fig. 23c), the DNN is distributed both according to neurons on the same layer (model parallelism), and according to the different layers (pipelining). Project Adam [Chilimbi et al. 2014] extends upon the ideas of DistBelief and exhibits the same types of parallelism. However, in Project Adam pipelining is restricted to different CPU cores on the same node.

6 DISTRIBUTED DEEP LEARNING

Distributing the training process among multiple nodes that do not share memory can take many forms. So far we have discussed training algorithms where there is only one copy of w, and its up-to-date value is directly visible to all processors. In distributed environments, there may be multiple instances of SGD (*training agents*) running independently, and thus the overall algorithm has to be adapted. Distribution schemes for deep learning can be categorized along three axes: **model consistency**, **parameter distribution**, and **distributed training**; where Fig. 24 and Table 8 summarize the applied techniques and optimizations.

6.1 Model Consistency

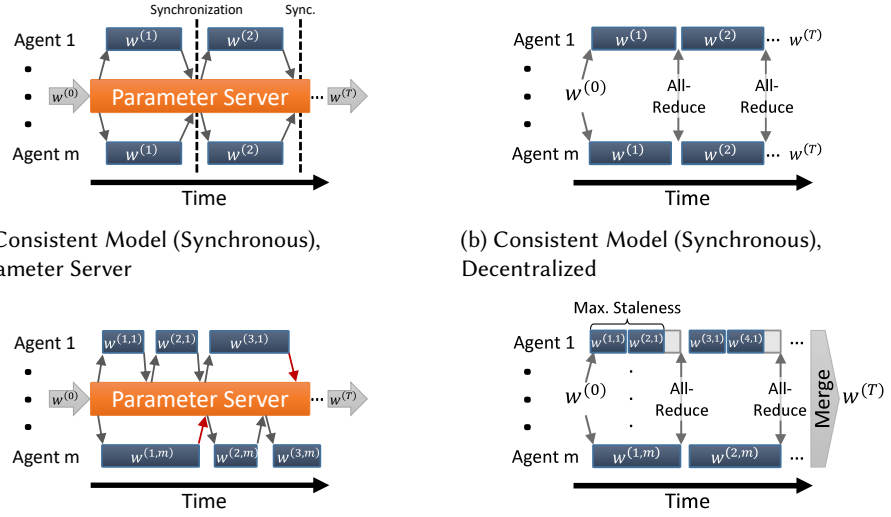
We denote training algorithms in which the up-to-date w is observed by everyone as *consistent model* methods (See Figures 25a and 25b). Directly dividing the computations among nodes creates a distributed form of data parallelism (Section 5), where all nodes have to communicate their updates to the others before fetching a new minibatch. To support distributed, data parallel SGD, we can modify Algorithm 2 by changing lines 3 and 7 to read (write) weights from (to) a parameter store,

Table 8. Overview of Distributed Deep Learning Methods

Category	Method	Papers
Model Consistency		
Synchronization	Synchronous	[Coates et al. 2013; Iandola et al. 2016a; Moritz et al. 2016; Seide et al. 2014a; Strom 2015; Yadan et al. 2013]
	Stale-Synchronous	[Gupta et al. 2016; Ho et al. 2013; Jiang et al. 2017; Lian et al. 2015; Zhang et al. 2016b]
	Asynchronous	[Dean et al. 2012; Keuper and Pfreundt 2015; Noel and Osindero 2014; Paine et al. 2013; Recht et al. 2011; Sa et al. 2015; Zhang et al. 2013]
	Nondeterministic Comm.	[Jin et al. 2016; Ram et al. 2009]
Parameter Distribution and Communication		
Centralization	Parameter Server (PS)	[Cui et al. 2016; Iandola et al. 2016a; Kim et al. 2016a; Li et al. 2014]
	Sharded PS	[Chilimbi et al. 2014; Dean et al. 2012; Jiang et al. 2017; Kurth et al. 2017; Le et al. 2012; Xing et al. 2015; Zhang et al. 2015, 2017]
	Hierarchical PS	[Gupta et al. 2016; Hsieh et al. 2017; Yu et al. 2016]
	Decentralized	[Jin et al. 2016; Lian et al. 2017]
Compression	Quantization	[Alistarh et al. 2017; Chen et al. 2014; Courbariaux and Bengio 2016; Courbariaux et al. 2015; Dettmers 2015; Gupta et al. 2015; Han et al. 2016; Hubara et al. 2016; Köster et al. 2017; Li and Liu 2016; Rastegari et al. 2016; Sa et al. 2015; Seide et al. 2014a; Wen et al. 2017; Zhou et al. 2016]
	Sparsification	[Aji and Heafield 2017; Chen et al. 2017; Dryden et al. 2016; Lin et al. 2018; Renggli et al. 2018; Shokri and Shmatikov 2015; Strom 2015]
	Other Methods	[Chollet 2016; Howard et al. 2017; Iandola et al. 2016b; Kim et al. 2016b; Li et al. 2017; Xie et al. 2016; Zhang et al. 2015]
Distributed Training		
Model Consolidation	Ensemble Learning	[Im et al. 2016; Lee et al. 2015; Simpson 2015]
	Knowledge Distillation	[Ba and Caruana 2014; Hinton et al. 2015]
	Model Averaging	Direct [Blanchard et al. 2017; Chen and Huo 2016; Miao et al. 2014; Zinkevich et al. 2010] Elastic Averaging [Jin et al. 2016; Lian et al. 2017; You et al. 2017a; Zhang et al. 2015] Natural Gradient [Povey et al. 2014]
Optimization Algorithms	Parameter Search	[Byrd et al. 2016; Chung et al. 2017; Dean et al. 2012; He et al. 2017; Johnson and Zhang 2013; Karras et al. 2017; Krishnan et al. 2018; Le et al. 2011; LeCun et al. 1998; Lorenzo et al. 2017; Martens 2010; Miikkulainen et al. 2017; Moritz et al. 2016; Petroski Such et al. 2017; Taylor et al. 2016; Verbancsics and Harguess 2015; Xie and Yuille 2017]
	Hyper-Parameter Search	[Baker et al. 2017b; Hadjis et al. 2016; Hazan et al. 2018; Jaderberg et al. 2017; Klein et al. 2016; Lorenzo et al. 2017; Miikkulainen et al. 2017; Snoek et al. 2012; Zhang et al. 2017]
	Architecture Search	[Baker et al. 2017a; Brock et al. 2017; Elsken et al. 2017; Liu et al. 2017; Liu et al. 2018b; Negrinho and Gordon 2017; Pham et al. 2018; Real et al. 2018; Real et al. 2017; Xie and Yuille 2017; Young et al. 2017; Zhong et al. 2017; Zoph and Le 2017; Zoph et al. 2017]

which may be centralized or decentralized (see Section 6.2). This incurs a substantial overhead on the overall system, which hinders training scaling.

Recent works relax the synchronization restriction, creating an *inconsistent model* (Fig. 25c). As a result, a training agent i at time t contains a copy of the weights, denoted as $w^{(\tau,i)}$ for $\tau \leq t$, where $t - \tau$ is called the *staleness* (or lag). A well-known instance of inconsistent SGD is the HOGWILD algorithm [Recht et al. 2011], which allows training agents to read and write parameters at will, overwriting the existing values. HOGWILD has been proven to converge for sparse learning problems, where updates only modify small subsets of w. Based on foundations of distributed asynchronous SGD [Tsitsiklis et al. 1986], the proof imposes Lipschitz continuous differentiability and strong convexity on f_w, and only converges as long as the staleness, i.e., the maximal number of iterations between reading and writing w, is bounded [Recht et al. 2011].



(a) Consistent Model (Synchronous), Parameter Server

(b) Consistent Model (Synchronous), Decentralized

(c) Inconsistent Model (Asynchronous), Parameter Server

(d) Inconsistent Model (Stale-Synchronous), Decentralized

Fig. 25. Training Distribution in Deep Learning

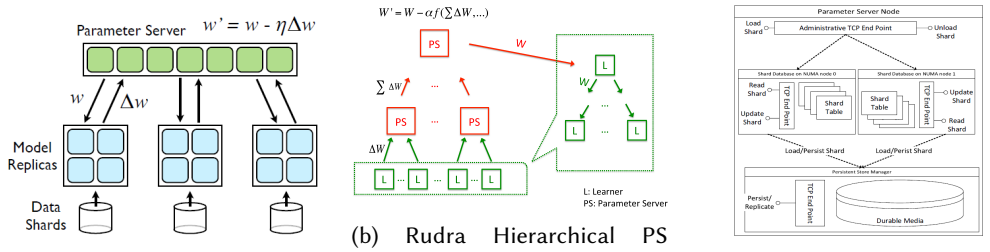
The HOGWILD algorithm was originally designed for shared-memory architectures, but has since been extended [Dean et al. 2012; Noel and Osindero 2014] to distributed-memory systems, in which it still attains convergence for deep learning problems. To mitigate the interference effect of overwriting w at each step, the implementation transfers the gradient ∇w instead of w from the training agents. Asymptotically, the lack of synchronization in HOGWILD and its gradient-communicating variants admits an optimal SGD convergence rate of $O(1/\sqrt{mT})$ for m participating nodes [Agarwal and Duchi 2011; Dekel et al. 2012; Lian et al. 2015], as well as linear scaling, as every agent can train almost independently.

To provide correctness guarantees in spite of asynchrony, Stale-Synchronous Parallelism (SSP) [Ho et al. 2013] proposes a compromise between consistent and inconsistent models. In SSP (Fig. 25d), the gradient staleness is enforced to be bounded by performing a global synchronization step after a maximal staleness may have been reached by one of the nodes. This approach works especially well in heterogeneous environments, where lagging agents (stragglers) are kept in check. To that end, distributed asynchronous processing has the additional advantage of adding and removing nodes on-the-fly, allowing users to add more resources, introduce node redundancy, and remove straggling nodes [Dean et al. 2012; PaddlePaddle 2017].

In practical implementations, the prominently-used model consistency approaches are synchronous for up to 32–50 nodes [Gibiansky 2017; Goyal et al. 2017], where the allreduce operation still scales nearly linearly; and asynchronous/SSP for larger clusters and heterogeneous environments [Hsieh et al. 2017; Jiang et al. 2017; Oyama et al. 2016; Zhang et al. 2016b].

6.2 Centralization

The choice between designing a centralized and a decentralized network architecture for DNN training depends on multiple factors [Lian et al. 2017], including the network topology, bandwidth, communication latency, parameter update frequency, and desired fault tolerance. A centralized



(a) DistBelief Sharded PS (adapted from [Dean et al. 2012])

(b) Rudra Hierarchical PS (adapted from [Gupta et al. 2016])

(c) Project Adam PS (adapted from [Chilimbi et al. 2014])

Fig. 26. Parameter Server Infrastructures

network architecture would typically include a *parameter server* (PS) infrastructure (e.g., Figures 25a, 25c, 26), which may consist of one or more specialized nodes; whereas a decentralized architecture (Figures 25b, 25d) would rely on allreduce to communicate parameter updates among the nodes.

The trade-off between using either distribution scheme can be modeled by the communication cost per global update. While the allreduce operation can be implemented efficiently for different message sizes and nodes (Section 2.5), the PS scheme requires each training agent to send and receive information to/from the PS nodes. Thus, not all network routes are used, and in terms of communication the operation is equivalent to a reduce-then-broadcast implementation of allreduce, taking T_{tree} time. However, the PS can keep track of a "global view" of the training procedure, performing gradient averaging in one place and enabling asynchronous operation of the training agents. This, in turn, allows nodes to communicate less information by performing some of the computations on the PS [Chilimbi et al. 2014], as well as increases fault tolerance by dynamic spin-up and removal of nodes during training.

The PS infrastructure is an abstract concept, and is not necessarily represented by one physical server. Sharded parameter servers [Chilimbi et al. 2014; Dean et al. 2012] divide the responsibility over w into multiple nodes, each containing a piece of the data. In conjunction with model parallelism and layer pipelining (Sections 5.2 and 5.3), this alleviates some of the congestion at the PS, as shown in Fig. 26a, in which each portion of a "model replica" (training agent) transmits its gradients and receives its weights from a different shard. Hierarchical parameter servers [Gupta et al. 2016; Yu et al. 2016] (Fig. 26b) further alleviate resource contention by assigning training agents with PS "leaves", propagating weights and gradients from specific agent groups up to the global parameter store. Rudra [Gupta et al. 2016] also studies the tradeoff in allowed staleness, number of agents, and minibatch size, showing that SSP performs better, but requires adapting the learning rate accordingly.

A PS infrastructure is not only beneficial for performance, but also for fault tolerance. The simplest form of fault tolerance in machine learning is checkpoint/restart, in which $w^{(t)}$ is periodically synchronized and persisted to a non-volatile data store (e.g., a hard drive). This is performed locally in popular deep learning frameworks, and globally in frameworks such as Poseidon [Zhang et al. 2017]. Besides checkpoints, fault tolerance in distributed deep learning has first been tackled by DistBelief [Dean et al. 2012; Le et al. 2012]. In the system, training resilience is increased by both introducing computational redundancy in the training agents (using different nodes that handle the same data), as well as replicating parameter server shards. In the former, an agent, which is constructed from multiple physical nodes in DistBelief via hybrid parallelism (Section 5.4), is assigned multiple times to separate groups of nodes. Allocating redundant agents enables

handling slow and faulty replicas ("stragglers") by cancelling their work upon completion of the faster counterpart. As for the latter resilience technique, in DistBelief and Project Adam [Chilimbi et al. 2014], the parameters on the PS are replicated and persisted on non-volatile memory using a dedicated manager, as can be seen in Fig. 26c. Project Adam further increases the resilience of distributed training by using separate communication endpoints for replication and using Paxos consensus between PS nodes.

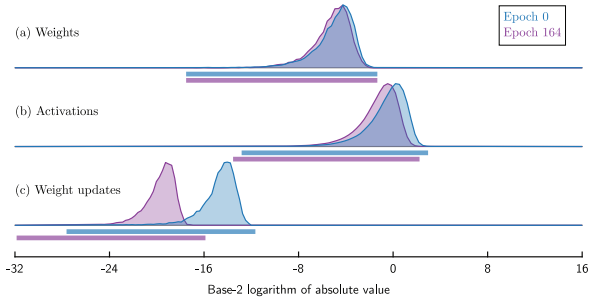
Applying weight updates in a distributed environment is another issue to be addressed. In Section 2.1, we establish that all popular weight rules are first-order with respect to the required gradients (Table 3). As such, both centralized and decentralized schemes can perform weight updates by storing the last gradient and parameter values. Since GPUs are commonly used when training DNNs (Fig. 4a), frameworks such as GeePS [Cui et al. 2016] implement a specialized PS for accelerator-based training agents. In particular, GeePS incorporates additional components over a general CPU PS, including CPU-GPU memory management components for weight updates.

In addition to reducing local (e.g., CPU-GPU) memory copies, PS infrastructures enable reducing the amount of information communicated over the network. In DistBelief, agents send $\nabla w^{(t,i)}$ to the PS as opposed to Hogwild [Recht et al. 2011], which transmits $w^{(t,i)}$ in both directions. This allows the PS to aggregate information from multiple agents, rather than overwriting it, and enables better compression (as we discuss in the next section). Project Adam extends this behavior, utilizing the fact that the PS is a compute-capable node to offload computation in favor of communicating less. In particular, Project Adam implements two different weight update protocols. For convolutional layers, in which the weights are sparse, gradients are communicated directly. However, in fully connected layers, the output of the previous layer $x \in X^{C_{in} \times N}$ and error $\frac{\partial \ell}{\partial y} \in X^{C_{out} \times N}$ are transmitted instead, and ∇w is computed on the PS (see Appendix A for computation). Therefore, with m nodes communication is modified from $m \cdot C_{out} \cdot C_{in}$ to $m \cdot N \cdot (C_{out} + C_{in})$, which may be significantly smaller, and balances the load between the agents and the normally under-utilized PS.

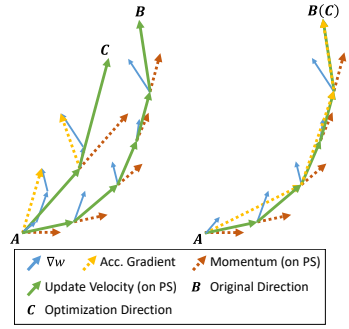
Dogwild [Noel and Osindero 2014] uses low-level unreliable communication for gradients (raw sockets), showing linear scaling in some cases.

Parameter servers also enable handling heterogeneity, both in training agents [Jiang et al. 2017] and in network settings (e.g., latency) [Hsieh et al. 2017]. The former work models distributed SGD over clusters with heterogeneous computing resources, and proposes two distributed algorithms based on stale-synchronous parallelism. Specifically, by decoupling global and local learning rates, unstable convergence caused by stragglers is mitigated. The latter work, Gaia [Hsieh et al. 2017], acknowledges the fact that training may be geo-distributed, i.e., originating from different locations, and proposes a hierarchical PS infrastructure that only synchronizes "significant" (large enough gradient) updates between data centers. To support this, the Approximate Synchronous Parallel model is defined, proven to retain convergence for SGD, and empirically shown to converge up to 5.6× faster with GoogLeNet than treating the system as homogeneous.

In a decentralized setting, load balancing can be achieved using asynchronous training. However, performing the parameter exchange cannot use the allreduce operation, as it incurs global synchronization. One approach to inconsistent decentralized parameter update is to use Gossip Algorithms [Boyd et al. 2005], in which each node communicates with a fixed number of random nodes (typically in the order of 3). With very high probability [Drezner and Barak 1986], after communicating for $1.639 \cdot log_2 m$ time-steps, where m is the number of nodes, the data will have been disseminated to the rest of the nodes. As strong consistency is not required for distributed deep learning, this method has shown marginal success for SGD [Jin et al. 2016; Ram et al. 2009], attaining both convergence and faster performance than allreduce SGD up to 32 nodes. On larger



(a) Distribution of Parameter and Gradient Values (ResNet on CIFAR-10, adapted from [Köster et al. 2017])



(b) Local Gradient Accumulation without (Left) and with (Right) Momentum Correction (adapted from [Lin et al. 2018])

Fig. 27. Parameter and Gradient Quantization

systems, the resulting test accuracy degrades. One approach to improve this could be to employ deterministic correction protocols [Hoefler et al. 2017].

6.3 Parameter and Gradient Compression

The distributed SGD algorithm requires all-to-all communication to operate correctly. As discussed above, reducing the *number* of messages (via an inconsistent view of w or efficient collective operations) is possible. Here, we discuss reducing the *size* of each message.

There are two general ways to conserve communication bandwidth in distributed deep learning: compressing the parameters with efficient data representations, and avoiding sending unnecessary information altogether, resulting in communication of sparse data structures. While the methods in the former category are orthogonal to the network infrastructure, the methods applied in the latter category differ when implemented using centralized (PS) and decentralized topologies.

6.3.1 Quantization. A prominent data representation for gradient (or parameter) compression is quantization, i.e., mapping continuous information into buckets that represent sets of values (usually ranges). It has been shown [Köster et al. 2017] that the distributions of parameter and gradient values are narrowly dispersed (Fig. 27a), thus these methods are effective in representing the working range to reduce the number of bits per parameter. This method has been successfully utilized in deep learning, both during training [Dettmers 2015; Gupta et al. 2015; Hubara et al. 2016] and for inference, where values are quantized post-training [Rastegari et al. 2016; Zhou et al. 2016]. Some papers go so far as to quantize gradients to binary [Courbariaux et al. 2015; Seide et al. 2014a] or ternary [Li and Liu 2016] values, while still attaining convergence with marginally reduced accuracy.

Quantization is commonly performed by way of reducing floating-point dynamic range [Dettmers 2015; Gupta et al. 2015; Sa et al. 2015]. In particular, such methods represent IEEE 754 32-bit floating-point values with fewer bits. While already applied to inference hardware [Chen et al. 2014], the first successful instance of reduced precision for training [Gupta et al. 2015] was performed with IEEE 754 16-bit float values ("half precision"). As evaluated in the paper, quantized training does not work "out-of-the-box" for lossy compression such as reduced precision. Rather, it depends on rounding the parameters in a way that preserves the expected value ($\mathbb{E}$) of the parameters. To

resolve this issue, the paper proposes Stochastic Rounding [Gupta et al. 2015], which randomly rounds numbers down or up, providing correct values in expectation.

Other forms of quantization extend upon these ideas. QSGD [Alistarh et al. 2017] generalizes stochastic rounding to stochastic quantization, and proposes multi-level gradient quantization schemes. Deep Compression [Han et al. 2016] also employs the lossless Huffman Coding [Huffman 1952] to further increase storage efficiency without impairing convergence. Binarized Neural Networks (BNNs) [Courbariaux and Bengio 2016], Ternary Weight Networks [Li and Liu 2016], TernGrad [Wen et al. 2017], and DoReFa-Net [Zhou et al. 2016] quantize networks to binary parameters, ternary parameters, ternary gradients, and binary parameters+ternary gradients, respectively. Both BNNs (in some cases) and TernGrad use stochastic rounding to lower the input representation. Lastly, FlexPoint [Köster et al. 2017] only computes the mantissa portion of the floating-point values as fixed-point math, sharing the exponent part among multiple parameters/gradients. To accommodate changes to the exponents (which the paper claims are relatively infrequent), predictive analysis is used for estimating subsequent values.

Essential to the convergence of SGD with lossy quantization is local gradient accumulation. Particularly in distributed environments, where the updates are inconsistent, it is important to carry the quantization error to the next gradient, accumulating error to avoid drift. The idea originates from Sigma-Delta Modulation [Seide et al. 2014a], and has proven to be successful in many cases. Deep Gradient Compression [Lin et al. 2018] extends this idea by correcting momentum as well (Fig. 27b), further decreasing the loss in accuracy to become non-negligible, and even resulting in a minor accuracy increase.

6.3.2 Sparsification. When training, DNNs (and CNNs in particular) exhibit sparse gradients during parameter updates. This is primarily due to the very large number of parameters that do not necessarily change all at once; and layers such as the convolutional layer, in which the optimization process may improve the accuracy of certain convolution kernels. Therefore, the full gradient is not necessary to retain convergence, and various methods that leverage this feature have been proposed.

The first application of gradient sparsification [Strom 2015] prunes gradient values using a static threshold, below which an element should not be sent. Results show up to 54× speedup for 80 nodes and even an up to 1.8% reduction in error. The authors achieved a compression ratio (which also includes 32-bit fixed point quantization) of 846–2,871× for a non-convolutional DNN. Subsequent works propose relative (e.g., top 1%) [Aji and Heafield 2017; Chen et al. 2017; Shokri and Shmatikov 2015] and adaptive thresholds [Dryden et al. 2016] to transmit only the "important" gradients, based on their absolute value. To counter the accuracy loss as a result of sparsification, some works suggest to condition gradient values by changing the DNN architecture, adding various normalization layers [Aji and Heafield 2017]; whereas others [Lin et al. 2018] propose local gradient clipping (Section 3.2) and warm-up training.

In a centralized setting (Section 6.2), distributing sparse gradients is straightforward — sparse messages are sent between the training agents and the PS. However, implementing the necessary allreduce in a decentralized setting is not as simple because each agent may contribute different non-zero indices (dimensions) in its gradient. Kylix [Zhao and Canny 2014] implements sparse allreduce in two steps, first exchanging the indices and then the data. While this is desirable for systems where the sparsity pattern per node does not change, in deep learning the gradient indices differ with each iteration. SparCML [Renggli et al. 2018] targets the specifics of deep learning explicitly by supporting arbitrarily changing indices in a framework for sparse allreduce operations. SparCML combines sending only the top-k most significant indices with quantization and supports sparse vectors of heterogeneous sizes. The system switches between a sparse and dense representation

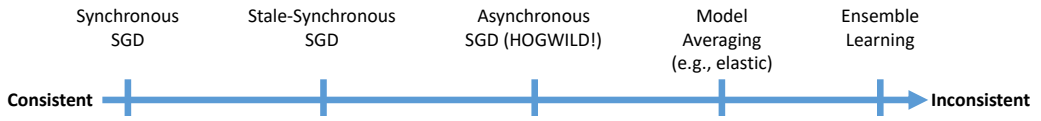


Fig. 28. Parameter Consistency Spectrum

automatically, informed by a simple performance model. SparCML achieves a speedup of more than 20× over a well-tuned CNTK implementation on Ethernet.

6.3.3 Other Techniques. In Section 6.2, we discuss Project Adam sending activations and errors instead of parameters, decreasing the overall footprint for fully connected layers in favor of redundant computation on the PS. The Poseidon (formerly Petuum) framework [Xing et al. 2015; Zhang et al. 2015, 2017] extends the idea of transmitting decomposed outer products $u \cdot v^T$ of w, generalizing the concept to other fields in machine learning as Sufficient Factor Broadcasting (SFB) [Xie et al. 2016]. With SFB, the activations are not sent to the PS, but rather broadcast between the training agents for local recomposition. SFB should work best in centralized topologies, as recomposing the gradients in a decentralized environment causes each agent to process $m - 1$ additional outer products with each step, where m is the number of agents. However, the authors claim [Xie et al. 2016] that the cost of recomposition is negligible compared to communication.

Since the decomposed weights are not additive, as opposed to gradients, SFB incurs all-to-all communication between training agents. To overcome scalability issues, the paper suggests partial broadcasting [Xie et al. 2016], where nodes communicate with a predetermined subset of the other nodes. By trading off gradient update latency ($\propto \mathbf{D}$) for bandwidth ($\propto \mathbf{W}$), the paper shows that convergence can still be attained, equating the delayed updates with stale gradients (Section 6.1).

A different approach to reduce DNN memory footprints is to design them specifically for that purpose [Chollet 2016; Iandola et al. 2016b; Kim et al. 2016b; Li et al. 2017]. Such works make use of memory-efficient layers and techniques, mostly applied to convolutions, to train networks that fit on devices such as FPGAs and mobile phones. The applied techniques include layers constructed from a multitude of 1×1 convolutions [Iandola et al. 2016b], reshaping [Li et al. 2017] or applying Tucker Decomposition [Kim et al. 2016b] on convolution tensors, and separable convolutions (sequential application of reduced-dimension convolutions) [Chollet 2016; Howard et al. 2017]. The papers show that DNNs can decrease in size (up to 50×) and evaluation time (up to 6.13×), exhibiting a minor reduction in accuracy.

6.4 Model Consolidation

In this section, we discuss the far (inconsistent) end of the parameter consistency spectrum (shown in Fig. 28) in distributed deep learning. In such cases, parameter updates are highly infrequent (or nonexistent), and thus precautions must be taken with the received values, or use different methods to consolidate the results.

In particular, rather than running data-parallel SGD on multiple nodes, distributed deep learning can be achieved by assigning training agents with different copies of w and combining the resulting models, either post-training or several times during training. While the latter can be seen as a generalization of an inconsistent view of w, the former may entirely change the training and inference processes, depending on the method.

6.4.1 Ensemble Learning and Knowledge Distillation. A widely-used technique for post-training consolidation is *ensemble learning* [Im et al. 2016; Lee et al. 2015; Simpson 2015]. With ensembles,

multiple instances of w are trained separately on the same dataset, and the overall prediction is the average of the predictions of the ensemble members, i.e., $f(x) = \frac{1}{m}\sum_{i=0}^{m} f_{w^{(T,i)}}(x)$. Ensemble learning has been used extensively in machine learning before the deep learning era [Dietterich 2000] as a form of boosting, and typically increases the overall accuracy over a single model. Thus, it is routinely applied in machine learning competitions such as ILSVRC [Deng et al. 2009] and in industrial applications. Distributed training of ensembles is a completely parallel process, requiring no communication between the agents. However, works such as TreeNets [Lee et al. 2015] (Section 5.2) combine ensemble learning with custom (ensemble-aware) loss functions to promote diversity between ensemble members.

Given that ensembles consume a factor of m more memory and compute power, another post-training model consolidation technique is to reduce the size of a DNN using *knowledge distillation* [Ba and Caruana 2014; Hinton et al. 2015]. In this scheme, training is performed in two steps: in the first step, a large network or an ensemble is trained normally; and the second step trains a single neural network to mimic the output of the large ensemble. Results [Hinton et al. 2015] show that the second network is easier to train on the ensemble than on a labeled dataset, attaining the same word error rate as an ensemble of 10 DNNs.

6.4.2 Model Averaging. Another technique for consolidating models is *model averaging* [Polyak and Juditsky 1992]. Such methods may separately run m SGD instances on different machines, aggregating the parameters only once (post-training) [Zinkevich et al. 2010] or every few iterations [Chen and Huo 2016; Miao et al. 2014]. While these methods are proven to converge, applying stale-synchronous SGD (Section 6.1) leads to higher overall accuracy.

To overcome accuracy degradation as a result of infrequent averaging, more sophisticated consolidation methods include Elastic Averaging SGD (EASGD) [Zhang et al. 2015] and Natural Gradient Descent [Povey et al. 2014]. EASGD is based on a centralized environment (i.e., including a PS), extending direct averaging by using elastic forces between the training agents' view of w ($w^{(t,i)}$) and the PS's view ($\bar{w}$). This allows the agents to "explore" further by increasing the possible distance of each agent from the average, and also allows to communicate sparsely with respect to time (iterations). EASGD was reported [Zhang et al. 2015] to outperform the DistBelief [Dean et al. 2012] Downpour SGD method in terms of accuracy, shown to be tolerant in terms of update delay, and was used successfully in practice for communication reduction by other works [Lian et al. 2017; You et al. 2017a].

Natural Gradient Descent (NG-SGD) can also be used to deal with diverging parameters in different agents [Povey et al. 2014]. NG-SGD modifies SGD to define learning rate matrices, approximating the inverse Fisher matrix and thus natural gradients. By averaging agent parameters only every k samples (typically in the order of hundreds of thousands), the algorithm allows agents to gradually diverge and synchronize less than traditional SGD. NG-SGD is successfully used in the Kaldi speech recognition toolkit [Povey et al. 2014], where the method converges both faster and with higher accuracy than SGD.

In distributed settings, algorithms are also inspected w.r.t. fault tolerance. Krum [Blanchard et al. 2017] is a Byzantine fault-tolerant [Lamport et al. 1982] SGD algorithm, allowing up to f Byzantine training agents. In particular, the paper shows that any gradient aggregation rule based on linear combination cannot sustain a single Byzantine agent. By combining specific $m - f$ gradients (that are closest to each other), Krum is able to overcome adversarial gradient inputs from the network.

6.5 Optimization Algorithms and Architecture Search

As training in deep learning is a nonlinear optimization (Section 2), other algorithms that exhibit concurrency can be used in SGD's stead [Bottou et al. 2016]. Furthermore, it is possible to use

excess computational power to perform meta-optimization, searching for better hyper-parameters and DNN architectures for a given problem.

6.5.1 Parameter Search. Supervised learning can either be viewed as a stochastic optimization process that uses one or a minibatch of samples at a time, or it can be expressed as a batch optimization problem, where the entire dataset is necessary to obtain gradients for descent. Batch optimization has been used for deep learning since the inception of DNNs [LeCun et al. 1998], using first- and second-order methods [Nocedal and Wright 2006] such as Levenberg-Marquardt, Conjugate Gradient (CG), and L-BFGS. Although considerably more computationally expensive than SGD, there are several advantages to such approaches, including increased concurrency (as data-parallelism increases) and better theoretical convergence guarantees (e.g., quadratic convergence for second-order methods). It is worth noting that large-minibatch training (Sections 2.6 and 5.1), which is gaining traction in recent years, represents a middle ground between SGD and batch methods. Such methods are beneficial both due to inherent concurrency, and due to stochasticity, which, despite the sublinear rate of convergence, works well in practice.

For distributed deep learning, batch methods [Le et al. 2011] (specifically CG and L-BFGS) and Hessian-free second-order optimization [Chung et al. 2017; He et al. 2017; Martens 2010] have initially been favored due to their apparent scalability compared to traditional SGD (Algorithm 1). However, following the successful DistBelief [Dean et al. 2012] implementation of both inconsistent SGD (called Downpour SGD, based on HOGWILD [Recht et al. 2011]) and distributed L-BFGS (called Sandblaster); it was found that the quadratic increase of batch methods in memory, communication, and computations due to high dimensionality is not desirable, as stochastic methods achieve the same results and scale better. To overcome this issue, stochastic variants of L-BFGS have been proposed [Byrd et al. 2016; Moritz et al. 2016] and proven to converge linearly [Moritz et al. 2016].

Other optimization algorithms applied to deep learning attempt to: (a) reduce the variance of SGD incurred by random sampling [Johnson and Zhang 2013], (b) use the Alternating Direction Method of Multipliers (ADMM) [Boyd et al. 2011] to skip the backpropagation altogether [Taylor et al. 2016], and (c) use the Neumann series expansion to approximate the Hessian matrix [Krishnan et al. 2018], scaling to large minibatch sizes (32k) with no accuracy loss or substantial computational overhead.

Gradient-free evolutionary algorithms have also been employed for deep learning and distributed deep learning, where examples include Genetic Algorithms [Petroski Such et al. 2017; Xie and Yuille 2017], Neuro-Evolution [Miikkulainen et al. 2017; Verbancsics and Harguess 2015], and Particle-Swarm Optimization [Lorenzo et al. 2017]. Apart from recombination/evolution steps, training behavior is similar to ensemble learning, and thus these algorithms are more amenable to parallelism than traditional gradient descent. As we show in the rest of this section, the gradient-independent nature of such algorithms enable their use for meta-optimization of both hyper-parameters and DNN architectures.

6.5.2 Hyper-Parameter Search. The multitude of hyper-parameters in SGD (e.g., learning rate, momentum, maximal staleness) and their adverse effect on the resulting accuracy hinders research efforts into new techniques in machine learning. Until recently, the prominent method for hyper-parameter search was to perform parameter sweeps (i.e., grid search over feasible ranges). Since this method increases the overall time exponentially with the number of hyper-parameters, its effectiveness is limited by the availability of computing power.

Several methods try to expand beyond simple parameter sweeps by making educated guesses and tuning hyper-parameters during training. In the former class, methods include Bayesian optimization [Snoek et al. 2012], predictive analysis of the learning curves (e.g., training error,

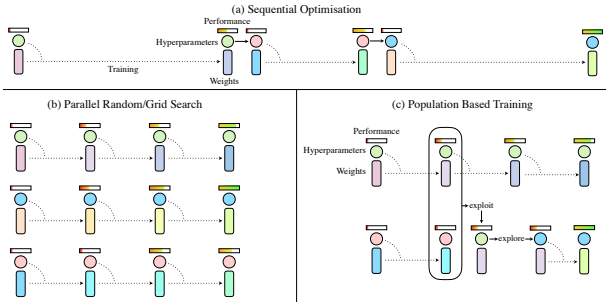


Fig. 29. Hyper-Parameter Sweep and Population-Based Training (adapted from [Jaderberg et al. 2017])

validation error) [Baker et al. 2017b; Klein et al. 2016] for dropping undesirable configurations, and sampling the hyper-parameter space efficiently using spectral methods such as Compressed Sensing [Hazan et al. 2018].

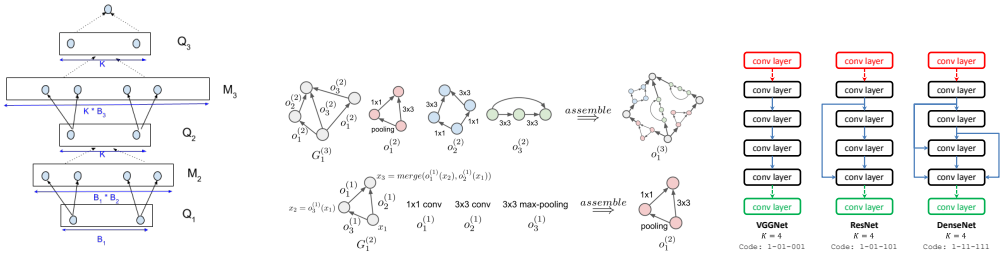
As for tuning hyper-parameters during training, Omnivore [Hadjis et al. 2016] employs predictive analysis and grid searches every predetermined number of minutes to modify the momentum and a hyper-parameter controlling local gradient staleness. The paper shows that in distributed environments, controlling the synchronous SGD node-group size during training can increase both accuracy and performance. YellowFin [Zhang et al. 2017] uses the local gradient curvature and variance to tune momentum, working especially well on LSTM-based models and asynchronous environments, performing up to 3.28× faster than the Adam optimizer (Table 3).

Metaheuristic optimization algorithms can inherently integrate hyper-parameter tuning with training, and are thus used for DNNs. Such methods include Particle Swarm Optimization (PSO) based deep learning [Lorenzo et al. 2017]; and CoDeepNEAT [Miikkulainen et al. 2017], a modification of the NEAT algorithm that simultaneously searches for hyper-parameter and architecture configurations (see below). Such methods scale almost linearly, due to abundance of independent computations.

Lastly, Population-Based Training [Jaderberg et al. 2017] (PBT, shown in Fig. 29) uses a reinforcement learning approach to "explore" and "exploit" the hyper-parameter space. In particular, each training agent independently samples (exploits) information from other agents every few SGD iterations. The information is then used to select the best configuration (e.g., using a t-test), and hyper-parameters are in turn perturbed (explored) to continue learning. This creates a decentralized topology where communication is nondeterministic (i.e., exploitation is performed with a randomly sampled agent), which may scale better as the number of training agents increases.

6.5.3 Architecture Search. Like feature engineering before the era of deep learning, manually crafting DNN architectures is naturally limited by human resourcefulness and creativity. This limitation promoted a recent rise of research into automated neural *architecture search*. Architecture search can be categorized into three distinct approaches: Discrete Search-Space Traversal (SST), Reinforcement Learning (RL), and Evolutionary Algorithms (EA).

Generally, SST-based search methods rely on defining a finite set of states to explore (e.g., a tree whose nodes' children are updated during the search process), and traversing those sets. As a result, concurrency in this search depends on the number of points in the search space at a given time. Examples of SST methods include DeepArchitect [Negrinho and Gordon 2017], which proposes a DNN definition language that allows programmers to explicitly define the space using options;



(a) PNASNet Search-Space Traversal (Q_n denotes cells with n blocks, adapted from [Liu et al. 2017])

(b) Hierarchical DNN Representation (adapted from [Liu et al. 2018b])

(c) Genetic Encoding of DNN Connections (adapted from [Xie and Yuille 2017])

Fig. 30. Methods for Automated Architecture Search

PNASNet [Liu et al. 2017], which searches for networks ordered by increasing complexity using a search algorithm based on A* (see Fig. 30a), conserving half the evaluated models compared to an equivalent RL approach [Zoph et al. 2017]; and SMASH [Brock et al. 2017], which assesses optimality (fitness) of candidate networks using another CNN that maps the given DNN architecture to weights for testing.

Many recent DNN architectures (Section 3.3) exhibit self-similarity and repeating sub-units (modules). This observation can be leveraged to dramatically reduce the number of explored architectures, composing networks hierarchically out of modules and basic blocks (e.g., convolution) as can be seen in Fig. 30b. This approach has been used successfully in the community, creating new candidates for both CNN modules [Liu et al. 2017; Liu et al. 2018b; Miikkulainen et al. 2017; Real et al. 2018; Zhong et al. 2017; Zoph et al. 2017] and RNN units [Pham et al. 2018; Zoph and Le 2017].

RL-based architecture search uses the accuracy of the resulting network as a reward function, whereas modifications to the DNN or its hyper-parameters are actions. In Neural Architecture Search (NAS) [Zoph and Le 2017], the parameters of each layer can be modified, but the number of layers is fixed. A sharded PS-based distributed system, in conjunction with policy gradient optimization [Williams 1992], is used for training. Other examples include MetaQNN [Baker et al. 2017a] and BlockQNN [Zhong et al. 2017], which operate similarly to NAS, but use Q-learning for optimization; and ENAS [Pham et al. 2018], which significantly reduces computational time over NAS (by three orders of magnitude) by sharing parameters across children DNNs (i.e., networks in the immediate search space).

Evolutionary Algorithms are advantageous for architecture search, as any function (not necessarily differentiable) can be optimized using these methods. HyperNEAT was the first EA successfully applied [Verbancsics and Harguess 2015] to deep learning, used for training weights and DNN architecture at the same time; and CoDeepNEAT [Miikkulainen et al. 2017] defines a variant of the NEAT algorithm to optimize hyper-parameters and architecture, using the self-similarity feature of DNNs by optimizing "blueprints" that are composed of modules. Genetic CNNs [Xie and Yuille 2017] uses Genetic Algorithms (GAs) by encoding the DNN connections as binary genes (as required in GAs, shown in Fig. 30c), and training the population of DNNs with every time-step, using the final accuracy as the fitness function. GAs are highly amenable to parallelism, and have been successfully used for very large-scale training [Young et al. 2017], where 18,000 nodes were used on the Titan

supercomputer for 24 hours to obtain state-of-the-art accuracy for segmentation and reconstruction problems.

Large-Scale Evolution [Real et al. 2017] also uses GAs, but defines a set of specific mutations (e.g., insert convolution, alter stride) that can be applied. Large-Scale Evolution outperforms some existing RL-based methods in terms of accuracy, as well as in terms of scalability, as GAs can run the entire population in parallel (where accuracy increases with population size in expectation). However, in the general case GA requires synchronous reductive communication between time-steps for selection of the fittest candidates. To overcome this issue, the paper employs *tournament selection* [Goldberg and Deb 1991], which only performs pairwise comparisons between population members.

Additional GA architecture search methods include the use of multi-level hierarchical representations of DNNs [Liu et al. 2018b] (Fig. 30b), which implement an asynchronous distributed tournament selection (centralized, queue-based implementation) with specialized mutation. Regularized Evolution (AmoebaNets) [Real et al. 2018] further extends GA with tournament selection by removing the oldest sample from the population each iteration (akin to death in nature), thus regularizing the optimization process. AmoebaNets outperform all existing methods, including manually engineered DNNs and RL-based searches, and are the current state-of-the-art in computer vision with 3.8% error for ImageNet and 2.13% error for CIFAR-10.

7 CONCLUDING REMARKS

The world of deep learning is brimming with concurrency. Nearly every aspect of training, from the computation of a convolution to the meta-optimization of DNN architectures, is inherently parallel. Even if an aspect is sequential, its consistency requirements can be reduced, due to the robustness of nonlinear optimization, to increase concurrency while still attaining reasonable accuracy, if not better. In this paper, we give an overview of many of these aspects, the respective approaches documented in literature, and provide concurrency analysis using the W-D model when possible.

It is hard to predict what the future holds for this highly active field of research (many have tried over the years). Engineering is advancing in a fast pace, producing systems that utilize increasing computational resources and DNN architectures that continually improve the state-of-the-art in various classification and regression problems. Below, we highlight potential directions for future research in parallel and distributed deep learning.

As research progresses, DNN architectures are becoming deeper and more interconnected, between consecutive and non-consecutive layers ("skip connections"). Apart from accuracy, considerable effort is devoted to reducing the memory footprint and number of operations [Howard et al. 2017; Real et al. 2018], in order to successfully run inference on mobile devices. This also means that post-training DNN compression [Han et al. 2016] will likely be researched further, and training compressible networks will be desirable. Since mobile hardware is limited in memory capacity and has to be energy efficient, specialized DNN computational hardware is frequently proposed [Sze et al. 2017]. We see this trend with the NVIDIA Tensor Cores [NVIDIA 2017b], the Tensor Processing Unit [Jouppi et al. 2017], other ASICs and FPGAs [Chen et al. 2014; Nurvitadhi et al. 2017], and even neuromorphic computing [Akopyan et al. 2015]. Handling DNN sparsity (e.g., after compression) is a focus for some ASICs [Zhang et al. 2016a], and advances in recurrent networks and attention learning [Chan et al. 2016; Xu et al. 2015] indicate that training and inference hardware would also need to work efficiently with variable-length inputs.

Computing individual layers is highly optimized today (see Section 4), and thus current research is oriented towards inter-layer and whole-DNN optimization. TensorFlow XLA [Google 2017] and Tensor Comprehensions [Vasilache et al. 2018] compile entire neural network graphs at once,

performing a variety of transformations (e.g., fusion) to optimize execution time, achieving 4× speedup over manually tuned individual layers. We expect research to continue in this direction to the point where DNN evaluation is close to optimal in terms of operations and shared-memory optimizations.

Techniques applied in distributed deep learning are converging to the point where a standard programming interface (or framework) can be designed. In the future, ecosystems such as Ease.ml [Li et al. 2017] may make the definition of a training scheme (e.g., with respect to centralization and gradient consistency) easier, hiding most of the low-level infrastructure setup. Combining the increasing support for cloud systems and elastic training [PaddlePaddle 2017] (where nodes can be spun up and removed at will) with the latest developments in evolutionary algorithms (see Section 6.5), we may see adaptive and financially-viable optimization methods rising to prominence.

Finally, deep learning is being used to solve increasingly complex problems such as routing algorithms [Graves et al. 2016] and hierarchical task combination [Frans et al. 2017]. Research towards Artificial General Intelligence is now focusing on multi-purpose networks [Johnson et al. 2016; Kaiser et al. 2017], which creates new, unexplored opportunities for model parallelism and different training algorithms. Searching for adequate multi-purpose networks may be beyond the ingenuity of a human team, and as meta-optimization (specifically, architecture search) and progressive training [Karras et al. 2017] increase in usability and quality; parameter sweeps and manual DNN architecture engineering will become obsolete. Supporting this claim is the fact that the current state-of-the-art CNN in computer vision [Real et al. 2018] (CIFAR-10 and ImageNet datasets) is the result of an automated architecture search.

ACKNOWLEDGMENTS

T.B.N. is funded by the ETH Postdoctoral Fellowship program.

REFERENCES

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, Dan 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. (2015). http://tensorflow.org/ Software available from tensorflow.org.

Alekh Agarwal and John C Duchi. 2011. Distributed Delayed Stochastic Optimization. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 873–881. http://papers.nips.cc/paper/4247-distributed-delayed-stochastic-optimization.pdf

Alham Fikri Aji and Kenneth Heafield. 2017. Sparse Communication for Distributed Gradient Descent. *CoRR* abs/1704.05021 (2017). arXiv:1704.05021 http://arxiv.org/abs/1704.05021

F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha. 2015. TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 10 (Oct 2015), 1537–1557. https://doi.org/10.1109/TCAD.2015.2474396

Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 1709–1720. http://papers.nips.cc/paper/6768-qsgd-communication-efficient-sgd-via-gradient-quantization-and-encoding.pdf

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 (Proceedings of Machine Learning Research)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. PMLR, New York, New York, USA, 173–182. http://proceedings.mlr.press/v48/amodei16.html

Jeremy Appleyard, Tomás Kociský, and Phil Blunsom. 2016. Optimizing Performance of Recurrent Neural Networks on GPUs. *CoRR* abs/1604.01946 (2016). arXiv:1604.01946 http://arxiv.org/abs/1604.01946

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*. ACM, New York, NY, USA, 119–129. https://doi.org/10.1145/277651.277678

Jimmy Ba and Rich Caruana. 2014. Do Deep Nets Really Need to be Deep? In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2654–2662. http://papers.nips.cc/paper/5484-do-deep-nets-really-need-to-be-deep.pdf

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. 2017a. Designing Neural Network Architectures using Reinforcement Learning. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1611.02167

Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. 2017b. Practical Neural Network Performance Prediction for Early Stopping. *CoRR* abs/1705.10823 (2017). arXiv:1705.10823 http://arxiv.org/abs/1705.10823

J Ballé, V Laparra, and E P Simoncelli. 2017. End-to-end optimized image compression. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1611.01704

R. Belli and T. Hoefler. 2015. Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS'15)*. IEEE.

Tal Ben-Nun, Ely Levy, Amnon Barak, and Eri Rubin. 2015. Memory Access Patterns: The Missing Piece of the Multi-GPU Puzzle. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, Article 19, 12 pages.

Yoshua Bengio. 2013. Deep Learning of Representations: Looking Forward. In *Statistical Language and Speech Processing: First International Conference, SLSP 2013, Tarragona, Spain, July 29-31, 2013. Proceedings*, Adrian-Horia Dediu, Carlos Martín-Vide, Ruslan Mitkov, and Bianca Truthe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–37. https://doi.org/10.1007/978-3-642-39593-2_1

Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. 2007. Greedy Layer-Wise Training of Deep Networks. In *Advances in Neural Information Processing Systems 19*, B. Schölkopf, J. C. Platt, and T. Hoffman (Eds.). MIT Press, 153–160. http://papers.nips.cc/paper/3048-greedy-layer-wise-training-of-deep-networks.pdf

Y. Bengio, P. Simard, and P. Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks* 5, 2 (Mar 1994), 157–166. https://doi.org/10.1109/72.279181

Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. 2017. Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 119–129. http://papers.nips.cc/paper/6617-machine-learning-with-adversaries-byzantine-tolerant-gradient-descent.pdf

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. https://doi.org/10.1145/324133.324234

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

L. Bottou, F. E. Curtis, and J. Nocedal. 2016. Optimization Methods for Large-Scale Machine Learning. *CoRR* abs/1606.04838 (2016). arXiv:1606.04838 http://arxiv.org/abs/1606.04838

S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. 2005. Gossip algorithms: design, analysis and applications. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, Vol. 3. 1653–1664 vol. 3. https://doi.org/10.1109/INFCOM.2005.1498447

Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. 2011. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Found. Trends Mach. Learn.* 3, 1 (Jan. 2011), 1–122. https://doi.org/10.1561/2200000016

Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (April 1974), 201–206. https://doi.org/10.1145/321812.321815

Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. 2017. SMASH: One-Shot Model Architecture Search through HyperNetworks. *CoRR* abs/1708.05344 (2017). arXiv:1708.05344 http://arxiv.org/abs/1708.05344

R. H. Byrd, S. L. Hansen, Jorge Nocedal, and Y. Singer. 2016. A Stochastic Quasi-Newton Method for Large-Scale Optimization. *SIAM Journal on Optimization* 26, 2 (2016), 1008–1031. https://doi.org/10.1137/140954362

Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. 2007. Collective Communication: Theory, Practice, and Experience: Research Articles. *Concurr. Comput. : Pract. Exper.* 19, 13 (Sept. 2007), 1749–1783. https://doi.org/10.1002/cpe.v19:13

W. Chan, N. Jaitly, Q. Le, and O. Vinyals. 2016. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 4960–4964. https://doi.org/10.1109/ICASSP.2016.7472621

Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High Performance Convolutional Neural Networks for Document Processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*, Guy Lorette (Ed.). Université de Rennes 1, Suvisoft, La Baule (France). https://hal.inria.fr/inria-00112631 http://www.suvisoft.com.

C.-Y. Chen, J. Choi, D. Brand, A. Agrawal, W. Zhang, and K. Gopalakrishnan. 2017. AdaComp : Adaptive Residual Gradient Compression for Data-Parallel Distributed Training. *CoRR* abs/1712.02679 (2017). arXiv:1712.02679 http://arxiv.org/abs/1712.02679

K. Chen and Q. Huo. 2016. Scalable training of deep learning machines by incremental block training with intra-block parallel optimization and blockwise model-update filtering. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 5880–5884. https://doi.org/10.1109/ICASSP.2016.7472805

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. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 269–284. https://doi.org/10.1145/2541940.2541967

Yunpeng Chen, Jianan Li, Huaxin Xiao, Xiaojie Jin, Shuicheng Yan, and Jiashi Feng. 2017. Dual Path Networks. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 4470–4478. http://papers.nips.cc/paper/7033-dual-path-networks.pdf

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 571–582. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi

Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734. http://www.aclweb.org/anthology/D14-1179

François Chollet. 2016. Xception: Deep Learning with Depthwise Separable Convolutions. *CoRR* abs/1610.02357 (2016). arXiv:1610.02357 http://arxiv.org/abs/1610.02357

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 *Advances in Neural Information Processing Systems 19*, B. Schölkopf, J. C. Platt, and T. Hoffman (Eds.). MIT Press, 281–288. http://papers.nips.cc/paper/3150-map-reduce-for-machine-learning-on-multicore.pdf

I. H. Chung, T. N. Sainath, B. Ramabhadran, M. Picheny, J. Gunnels, V. Austel, U. Chauhari, and B. Kingsbury. 2017. Parallel Deep Neural Network Training for Big Data on Blue Gene/Q. *IEEE Transactions on Parallel and Distributed Systems* 28, 6 (June 2017), 1703–1714. https://doi.org/10.1109/TPDS.2016.2626289

Dan C. Cireşan, Alessandro Giusti, Luca M. Gambardella, and Jürgen Schmidhuber. 2013. Mitosis Detection in Breast Cancer Histology Images with Deep Neural Networks. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2013*, Kensaku Mori, Ichiro Sakuma, Yoshinobu Sato, Christian Barillot, and Nassir Navab (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 411–418.

Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, and Bryan Catanzaro. 2013. Deep Learning with COTS HPC Systems. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28 (ICML'13)*. JMLR.org, III–1337–III–1345.

N. Cohen, O. Sharir, and A. Shashua. 2016a. Deep SimNets. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Vol. 00. 4782–4791. https://doi.org/10.1109/CVPR.2016.517

Nadav Cohen, Or Sharir, and Amnon Shashua. 2016b. On the Expressive Power of Deep Learning: A Tensor Analysis. In *29th Annual Conference on Learning Theory (Proceedings of Machine Learning Research)*, Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir (Eds.), Vol. 49. PMLR, Columbia University, New York, New York, USA, 698–728. http:

//proceedings.mlr.press/v49/cohen16.html

Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*.

Jason Cong and Bingjun Xiao. 2014. Minimizing Computation in Convolutional Neural Networks. In *Artificial Neural Networks and Machine Learning – ICANN 2014: 24th International Conference on Artificial Neural Networks, Hamburg, Germany, September 15-19, 2014. Proceedings*, Stefan Wermter, Cornelius Weber, Włodzisław Duch, Timo Honkela, Petia Koprinkova-Hristova, Sven Magg, Günther Palm, and Alessandro E. P. Villa (Eds.). Springer International Publishing, Cham, 281–290. https://doi.org/10.1007/978-3-319-11179-7_36

Matthieu Courbariaux and Yoshua Bengio. 2016. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR* abs/1602.02830 (2016). arXiv:1602.02830 http://arxiv.org/abs/1602.02830

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'15)*. MIT Press, Cambridge, MA, USA, 3123–3131. http://dl.acm.org/citation.cfm?id=2969442.2969588

Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. 2016. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 4, 16 pages. https://doi.org/10.1145/2901318.2901323

David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '93)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/155332.155333

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 - Volume 1 (NIPS'12)*. Curran Associates Inc., USA, 1223–1231. http://dl.acm.org/citation.cfm?id=2999134.2999271

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. 2012. Optimal Distributed Online Prediction Using Mini-batches. *J. Mach. Learn. Res.* 13, 1 (Jan. 2012), 165–202. http://dl.acm.org/citation.cfm?id=2503308.2188391

Olivier Delalleau and Yoshua Bengio. 2011. Shallow vs. Deep Sum-Product Networks. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 666–674. http://papers.nips.cc/paper/4350-shallow-vs-deep-sum-product-networks.pdf

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

L. Deng, D. Yu, and J. Platt. 2012. Scalable stacking and learning for building deep architectures. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2133–2136. https://doi.org/10.1109/ICASSP.2012.6288333

Tim Dettmers. 2015. 8-Bit Approximations for Parallelism in Deep Learning. *CoRR* abs/1511.04561 (2015). arXiv:1511.04561 http://arxiv.org/abs/1511.04561

Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent RNNs: Stashing Recurrent Weights On-Chip. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. PMLR, New York, New York, USA, 2024–2033. http://proceedings.mlr.press/v48/diamos16.html

Thomas G. Dietterich. 2000. Ensemble Methods in Machine Learning. In *Proceedings of the First International Workshop on Multiple Classifier Systems (MCS '00)*. Springer-Verlag, London, UK, UK, 1–15. http://dl.acm.org/citation.cfm?id=648054.743935

Zvi Drezner and Amnon Barak. 1986. An asynchronous algorithm for scattering information between the active nodes of a multicomputer system. *J. Parallel and Distrib. Comput.* 3, 3 (1986), 344 – 351. https://doi.org/10.1016/0743-7315(86)90020-1

N. Dryden, T. Moon, S. A. Jacobs, and B. V. Essen. 2016. Communication Quantization for Data-Parallel Training of Deep Neural Networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*. 1–8. https://doi.org/10.1109/MLHPC.2016.004

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. http://dl.acm.org/citation.cfm?id=1953048.2021068

V. Dumoulin and F. Visin. 2016. A guide to convolution arithmetic for deep learning. *CoRR* abs/1603.07285 (2016). arXiv:1603.07285 http://arxiv.org/abs/1603.07285

Jeffrey L. Elman. 1990. Finding Structure in Time. *Cognitive Science* 14, 2 (1990), 179–211. https://doi.org/10.1207/s15516709cog1402_1

T. Elsken, J.-H. Metzen, and F. Hutter. 2017. Simple And Efficient Architecture Search for Convolutional Neural Networks. *CoRR* abs/1711.04528 (2017). arXiv:1711.04528 http://arxiv.org/abs/1711.04528

Ludvig Ericson and Rendani Mbuvha. 2017. On the Performance of Network Parallel Training in Artificial Neural Networks. *CoRR* abs/1701.05130 (2017). arXiv:1701.05130 http://arxiv.org/abs/1701.05130

P. Farber and K. Asanovic. 1997. Parallel neural network training on Multi-Spert. In *Proceedings of 3rd International Conference on Algorithms and Architectures for Parallel Processing*. 659–666. https://doi.org/10.1109/ICAPP.1997.651531

Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. 2017. Meta Learning Shared Hierarchies. *CoRR* abs/1710.09767 (2017). arXiv:1710.09767 http://arxiv.org/abs/1710.09767

Alex Gaunt, Matthew Johnson, Maik Riechert, Daniel Tarlow, Ryota Tomioka, Dimitrios Vytiniotis, and Sam Webster. 2017. AMPNet: Asynchronous Model-Parallel Training for Dynamic Neural Networks. *CoRR* abs/1705.09786 (2017). arXiv:1705.09786 http://arxiv.org/abs/1705.09786

Andrew Gibiansky. 2017. Bringing HPC Techniques to Deep Learning. (2017). http://research.baidu.com/bringing-hpc-techniques-deep-learning/ [Online].

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research)*, Yee Whye Teh and Mike Titterington (Eds.), Vol. 9. PMLR, Chia Laguna Resort, Sardinia, Italy, 249–256.

David E. Goldberg and Kalyanmoy Deb. 1991. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. Foundations of Genetic Algorithms, Vol. 1. Elsevier, 69 – 93. https://doi.org/10.1016/B978-0-08-050684-5.50008-2

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2672–2680.

Google. 2017. TensorFlow XLA Overview. (2017). https://www.tensorflow.org/performance/xla

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). http://arxiv.org/abs/1706.02677

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538, 7626 (2016), 471–476.

W. Gropp, T. Hoefler, R. Thakur, and E. Lusk. 2014. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press.

Audrunas Gruslys, Remi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-Efficient Backpropagation Through Time. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 4125–4133. http://papers.nips.cc/paper/6221-memory-efficient-backpropagation-through-time.pdf

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Francis Bach and David Blei (Eds.), Vol. 37. PMLR, Lille, France, 1737–1746.

S. Gupta, W. Zhang, and F. Wang. 2016. Model Accuracy and Runtime Tradeoff in Distributed Deep Learning: A Systematic Study. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. 171–180. https://doi.org/10.1109/ICDM.2016.0028

Stefan Hadjis, Ce Zhang, Ioannis Mitliagkas, and Christopher Ré. 2016. Omnivore: An Optimizer for Multi-device Deep Learning on CPUs and GPUs. *CoRR* abs/1606.04487 (2016). arXiv:1606.04487 http://arxiv.org/abs/1606.04487

Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations (ICLR)* (2016). http://arxiv.org/abs/1510.00149

Elad Hazan, Adam Klivans, and Yang Yuan. 2018. Hyperparameter optimization: a spectral approach. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1706.00764

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV) (ICCV '15)*. IEEE Computer Society, Washington, DC, USA, 1026–1034. https://doi.org/10.1109/ICCV.2015.123

K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

Xi He, Dheevatsa Mudigere, Mikhail Smelyanskiy, and Martin Takac. 2017. Distributed Hessian-Free Optimization for Deep Neural Network. (2017). https://aaai.org/ocs/index.php/WS/AAAIW17/paper/view/15124

Geoffrey Hinton. 2012. Neural Networks for Machine Learning - Lecture 6a: Overview of Mini-batch Gradient Descent. (2012). http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the Knowledge in a Neural Network. In *NIPS Deep Learning and Representation Learning Workshop*. http://arxiv.org/abs/1503.02531

G. E. Hinton, S. Osindero, and Y. W. Teh. 2006. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation* 18, 7 (July 2006), 1527–1554. https://doi.org/10.1162/neco.2006.18.7.1527

Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'13)*. Curran Associates Inc., USA, 1223–1231.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

T. Hoefler, A. Barak, A. Shiloh, and Z. Drezner. 2017. Corrected Gossip Algorithms for Fast Reliable Broadcast on Unreliable Systems. In *Proceedings of the 31st IEEE International Parallel & Distributed Processing Symposium (IPDPS'17)*. IEEE.

T. Hoefler and D. Moor. 2014. Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations. *Journal of Supercomputing Frontiers and Innovations* 1, 2 (Oct. 2014), 58–75.

T. Hoefler and T. Schneider. 2012. Optimization Principles for Collective Neighborhood Communications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 98:1–98:10.

T. Hoefler and M. Snir. 2011. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11)*. ACM, 75–85.

T. Hoefler and J. L. Traeff. 2009. Sparse Collective Operations for MPI. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop*.

Elad Hoffer, Itay Hubara, and Daniel Soudry. 2017. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 1729–1739. http://papers.nips.cc/paper/6770-train-longer-generalize-better-closing-the-generalization-gap-in-large-batch-training-of-neural-networks.pdf

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

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 Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, Berkeley, CA, USA, 629–647.

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *CoRR* abs/1609.07061 (2016). arXiv:1609.07061 http://arxiv.org/abs/1609.07061

D. A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (Sept 1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016b. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). arXiv:1602.07360 http://arxiv.org/abs/1602.07360

Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, and Kurt Keutzer. 2016a. FireCaffe: Near-Linear Acceleration of Deep Neural Network Training on Compute Clusters. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Paolo Ienne. 1993. *Architectures for Neuro-Computers: Review and Performance Evaluation*. Technical Report. EPFL, Lausanne, Switzerland.

Daniel Jiwoong Im, He Ma, Chris Dongjoo Kim, and Graham W. Taylor. 2016. Generative Adversarial Parallelization. *CoRR* abs/1612.04021 (2016). arXiv:1612.04021 http://arxiv.org/abs/1612.04021

Intel. 2009. *Intel Math Kernel Library. Reference Manual*. Intel Corporation.

Intel. 2017. MKL-DNN. (2017). https://01.org/mkl-dnn

Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 448–456.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Population Based Training

of Neural Networks. *CoRR* abs/1711.09846 (2017). arXiv:1711.09846 http://arxiv.org/abs/1711.09846

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 ACM International Conference on Multimedia*. 675–678.

Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware Distributed Parameter Servers. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 463–478. https://doi.org/10.1145/3035918.3035933

Peter H. Jin, Qiaochu Yuan, Forrest N. Iandola, and Kurt Keutzer. 2016. How to scale distributed deep learning? *ML Systems Workshop at NIPS* (2016).

Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda B. Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation. *CoRR* abs/1611.04558 (2016). arXiv:1611.04558 http://arxiv.org/abs/1611.04558

Rie Johnson and Tong Zhang. 2013. Accelerating Stochastic Gradient Descent using Predictive Variance Reduction. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 315–323. http://papers.nips.cc/paper/4937-accelerating-stochastic-gradient-descent-using-predictive-variance-reduction.pdf

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

Lukasz Kaiser, Aidan N. Gomez, Noam Shazeer, Ashish Vaswani, Niki Parmar, Llion Jones, and Jakob Uszkoreit. 2017. One Model To Learn Them All. *CoRR* abs/1706.05137 (2017). arXiv:1706.05137 http://arxiv.org/abs/1706.05137

Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2017. Progressive Growing of GANs for Improved Quality, Stability, and Variation. *CoRR* abs/1710.10196 (2017). arXiv:1710.10196 http://arxiv.org/abs/1710.10196

Janis Keuper and Franz-Josef Pfreundt. 2015. Asynchronous Parallel Stochastic Gradient Descent: A Numeric Core for Scalable Distributed Machine Learning Algorithms. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments (MLHPC '15)*. ACM, New York, NY, USA, Article 1, 11 pages. https://doi.org/10.1145/2834892.2834893

Hanjoo Kim, Jaehong Park, Jaehee Jang, and Sungroh Yoon. 2016a. DeepSpark: Spark-Based Deep Learning Supporting Asynchronous Updates and Caffe Compatibility. *CoRR* abs/1602.08191 (2016). arXiv:1602.08191 http://arxiv.org/abs/1602.08191

Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2016b. Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1511.06530

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1412.6980

Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. 2016. Learning curve prediction with Bayesian neural networks. In *International Conference on Learning Representations (ICLR)*.

Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J Pai, and Naveen Rao. 2017. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 1740–1750. http://papers.nips.cc/paper/6771-flexpoint-an-adaptive-numerical-format-for-efficient-training-of-deep-neural-networks.pdf

Shankar Krishnan, Ying Xiao, and Rif A. Saurous. 2018. Neumann Optimizer: A Practical Optimization Algorithm for Deep Neural Networks. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1712.03298

Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Master's thesis. http://www.cs.toronto.edu/~

Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *CoRR* abs/1404.5997 (2014). http://arxiv.org/abs/1404.5997

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., USA, 1097–1105.

Thorsten Kurth, Jian Zhang, Nadathur Satish, Evan Racah, Ioannis Mitliagkas, Md. Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, Jack Deslippe, Mikhail Shiryaev, Srinivas Sridharan, Prabhat, and Pradeep Dubey. 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 (SC '17)*. ACM, New York, NY, USA, Article 7, 11 pages. https://doi.org/10.1145/3126908.3126916

Griffin Lacey, Graham W. Taylor, and Shawki Areibi. 2016. Deep Learning on FPGAs: Past, Present, and Future. *CoRR* abs/1602.04283 (2016). arXiv:1602.04283 http://arxiv.org/abs/1602.04283

Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 382–401. https://doi.org/10.1145/357172.357176

Andrew Lavin and Scott Gray. 2016. Fast Algorithms for Convolutional Neural Networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Quoc V. Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y. Ng. 2011. On Optimization Methods for Deep Learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning (ICML'11)*. Omnipress, USA, 265–272. http://dl.acm.org/citation.cfm?id=3104482.3104516

Quoc V. Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg S. Corrado, Jeff Dean, and Andrew Y. Ng. 2012. Building High-level Features Using Large Scale Unsupervised Learning. In *Proceedings of the 29th International Coference on International Conference on Machine Learning (ICML'12)*. Omnipress, USA, 507–514.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444. https://doi.org/10.1038/nature14539

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Comput.* 1, 4 (Dec. 1989), 541–551. https://doi.org/10.1162/neco.1989.1.4.541

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

Yann LeCun and Corinna Cortes. 1998. The MNIST database of handwritten digits. (1998). http://yann.lecun.com/exdb/mnist.

Stefan Lee, Senthil Purushwalkam, Michael Cogswell, David J. Crandall, and Dhruv Batra. 2015. Why M Heads are Better than One: Training a Diverse Ensemble of Deep Networks. *CoRR* abs/1511.06314 (2015). http://arxiv.org/abs/1511.06314

Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. 2016. Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 54, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014977

Dawei Li, Xiaolong Wang, and Deguang Kong. 2017. DeepRebirth: Accelerating Deep Neural Network Execution on Mobile Devices. *CoRR* abs/1708.04728 (2017). arXiv:1708.04728 http://arxiv.org/abs/1708.04728

Fengfu Li and Bin Liu. 2016. Ternary Weight Networks. *CoRR* abs/1605.04711 (2016). arXiv:1605.04711 http://arxiv.org/abs/1605.04711

Mu Li, David G. Andersen, 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 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 583–598.

Tian Li, Jie Zhong, Ji Liu, Wentao Wu, and Ce Zhang. 2017. Ease.ml: Towards Multi-tenant Resource Sharing for Machine Learning Workloads. *CoRR* abs/1708.07308 (2017). arXiv:1708.07308 http://arxiv.org/abs/1708.07308

Yuxi Li. 2017. Deep Reinforcement Learning: An Overview. *CoRR* abs/1701.07274 (2017). arXiv:1701.07274 http://arxiv.org/abs/1701.07274

Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. 2015. Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'15)*. MIT Press, Cambridge, MA, USA, 2737–2745. http://dl.acm.org/citation.cfm?id=2969442.2969545

Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. 2017. Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 5336–5346.

Min Lin, Qiang Chen, and Shuicheng Yan. 2014. Network In Network. *International Conference on Learning Representations (ICLR)* (2014). http://arxiv.org/abs/1312.4400

Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally. 2018. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *Proceedings of the International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1712.01887

C. Liu, B. Zoph, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. 2017. Progressive Neural Architecture Search. *CoRR* abs/1712.00559 (2017). arXiv:1712.00559 http://arxiv.org/abs/1712.00559

Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. 2018b. Hierarchical Representations for Efficient Architecture Search. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1711.00436

Xingyu Liu, Jeff Pool, Song Han, and William J. Dally. 2018a. Efficient Sparse-Winograd Convolutional Neural Networks. *International Conference on Learning Representations (ICLR)* (2018). https://arxiv.org/abs/1802.06367

Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully Convolutional Networks for Semantic Segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Pablo Ribalta Lorenzo, Jakub Nalepa, Luciano Sanchez Ramos, and José Ranilla Pastor. 2017. Hyper-parameter Selection in Deep Neural Networks Using Parallel Particle Swarm Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. ACM, New York, NY, USA, 1864–1871. https://doi.org/10.1145/3067695.3084211

James Martens. 2010. Deep Learning via Hessian-free Optimization. In *Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10)*. Omnipress, USA, 735–742. http://dl.acm.org/citation.cfm?id=3104322.3104416

Michaël Mathieu, Mikael Henaff, and Yann LeCun. 2014. Fast Training of Convolutional Networks through FFTs. *International Conference on Learning Representations (ICLR)* (2014). http://arxiv.org/abs/1312.5851

Message Passing Interface Forum. 2015. MPI: A Message-Passing Interface Standard Version 3.1. (June 2015).

Yajie Miao, Hao Zhang, and Florian Metze. 2014. Distributed learning of multilingual DNN feature extractors using GPUs. In *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014*. 830–834.

Risto Miikkulainen, Jason Zhi Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. 2017. Evolving Deep Neural Networks. *CoRR* abs/1703.00548 (2017). arXiv:1703.00548 http://arxiv.org/abs/1703.00548

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. PMLR, New York, New York, USA, 1928–1937. http://proceedings.mlr.press/v48/mniha16.html

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (26 02 2015), 529–533. http://dx.doi.org/10.1038/nature14236

Philipp Moritz, Robert Nishihara, and Michael Jordan. 2016. A Linearly-Convergent Stochastic L-BFGS Algorithm. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research)*, Arthur Gretton and Christian C. Robert (Eds.), Vol. 51. PMLR, Cadiz, Spain, 249–258. http://proceedings.mlr.press/v51/moritz16.html

P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. 2016. SparkNet: Training Deep Networks in Spark. In *Proceedings of the International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1511.06051

U. A. Muller and A. Gunzinger. 1994. Neural net simulation on parallel computers. In *Neural Networks, 1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on*, Vol. 6. 3961–3966 vol.6. https://doi.org/10.1109/ICNN.1994.374845

Maryam M. Najafabadi, Flavio Villanustre, Taghi M. Khoshgoftaar, Naeem Seliya, Randall Wald, and Edin Muharemagic. 2015. Deep learning applications and challenges in big data analytics. *Journal of Big Data* 2, 1 (24 Feb 2015), 1. https://doi.org/10.1186/s40537-014-0007-7

R. Negrinho and G. Gordon. 2017. DeepArchitect: Automatically Designing and Training Deep Architectures. *CoRR* abs/1704.08792 (2017). arXiv:1704.08792 http://arxiv.org/abs/1704.08792

A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. 2009. Robust Stochastic Approximation Approach to Stochastic Programming. *SIAM Journal on Optimization* 19, 4 (2009), 1574–1609. https://doi.org/10.1137/070704277

Yurii Nesterov. 1983. A method of solving a convex programming problem with convergence rate O (1/k2). *Soviet Mathematics Doklady* 269 (1983), 543–547.

Netlib. 2017. Basic Linear Algebra Subprograms (BLAS). (2017). http://www.netlib.org/blas

Jiquan Ngiam, Zhenghao Chen, Daniel Chia, Pang W. Koh, Quoc V. Le, and Andrew Y. Ng. 2010. Tiled convolutional neural networks. In *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta (Eds.). Curran Associates, Inc., 1279–1287. http://papers.nips.cc/paper/4136-tiled-convolutional-neural-networks.pdf

J. Nocedal and S. Wright. 2006. *Numerical Optimization*. Springer New York.

Cyprien Noel and Simon Osindero. 2014. Dogwild!-Distributed Hogwild for CPU & GPU. In *NIPS Workshop on Distributed Machine Learning and Matrix Computations*.

Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. 2017. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/3020078.3021740

NVIDIA. 2017a. CUBLAS Library Documentation. (2017). http://docs.nvidia.com/cuda/cublas

NVIDIA. 2017b. Programming Tensor Cores in CUDA 9. (2017). https://devblogs.nvidia.com/programming-tensor-cores-cuda-9

Christopher Olah. 2015. Understanding LSTM Networks. (2015). http://colah.github.io/posts/2015-08-Understanding-LSTMs

Y. Oyama, A. Nomura, I. Sato, H. Nishimura, Y. Tamatsu, and S. Matsuoka. 2016. Predicting statistics of asynchronous SGD parameters for a large-scale distributed deep learning system on GPU supercomputers. In *2016 IEEE International Conference on Big Data (Big Data)*. 66–75. https://doi.org/10.1109/BigData.2016.7840590

PaddlePaddle. 2017. Elastic Deep Learning. (2017). https://github.com/PaddlePaddle/cloud/tree/develop/doc/edl

Thomas Paine, Hailin Jin, Jianchao Yang, Zhe Lin, and Thomas S. Huang. 2013. GPU Asynchronous Stochastic Gradient Descent to Speed Up Neural Network Training. *CoRR* abs/1312.6186 (2013). arXiv:1312.6186 http://arxiv.org/abs/1312.6186

S. J. Pan and Q. Yang. 2010. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering* 22, 10 (Oct 2010), 1345–1359. https://doi.org/10.1109/TKDE.2009.191

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the Difficulty of Training Recurrent Neural Networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28 (ICML'13)*. JMLR.org, III–1310–III–1318. http://dl.acm.org/citation.cfm?id=3042817.3043083

F. Petroski Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune. 2017. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *CoRR* abs/1712.06567 (2017). arXiv:1712.06567 http://arxiv.org/abs/1712.06567

H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. 2018. Efficient Neural Architecture Search via Parameter Sharing. *CoRR* abs/1802.03268 (2018). arXiv:1802.03268 http://arxiv.org/abs/1802.03268

B. T. Polyak and A. B. Juditsky. 1992. Acceleration of Stochastic Approximation by Averaging. *SIAM J. Control Optim.* 30, 4 (July 1992), 838–855. https://doi.org/10.1137/0330046

Daniel Povey, Xiaohui Zhang, and Sanjeev Khudanpur. 2014. Parallel training of Deep Neural Networks with Natural Gradient and Parameter Averaging. *CoRR* abs/1410.7455 (2014). arXiv:1410.7455 http://arxiv.org/abs/1410.7455

Hang Qi, Evan R. Sparks, and Ameet Talwalkar. 2017. Paleo: A Performance Model for Deep Neural Networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Ning Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural Networks* 12, 1 (1999), 145 – 151. https://doi.org/10.1016/S0893-6080(98)00116-6

Rolf Rabenseifner. 2004. Optimization of collective reduction operations. In *International Conference on Computational Science*. Springer, 1–9.

Ali Rahimi and Benjamin Recht. 2017. Reflections on Random Kitchen Sinks. (2017). http://www.argmin.net/2017/12/05/kitchen-sinks NIPS 2017 Test of Time Award Talk.

Rajat Raina, Anand Madhavan, and Andrew Y. Ng. 2009. Large-scale Deep Unsupervised Learning Using Graphics Processors. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09)*. ACM, New York, NY, USA, 873–880. https://doi.org/10.1145/1553374.1553486

S. Sundhar Ram, A. Nedic, and V. V. Veeravalli. 2009. Asynchronous gossip algorithms for stochastic optimization. In *2009 International Conference on Game Theory for Networks*. 80–81. https://doi.org/10.1109/GAMENETS.2009.5137386

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *CoRR* abs/1603.05279 (2016). arXiv:1603.05279 http://arxiv.org/abs/1603.05279

E. Real, A. Aggarwal, Y. Huang, and Q. V Le. 2018. Regularized Evolution for Image Classifier Architecture Search. *CoRR* abs/1802.01548 (2018). arXiv:1802.01548 http://arxiv.org/abs/1802.01548

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V. Le, and Alexey Kurakin. 2017. Large-Scale Evolution of Image Classifiers. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 2902–2911. http://proceedings.mlr.press/v70/real17a.html

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 693–701. http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf

C. Renggli, D. Alistarh, and T. Hoefler. 2018. SparCML: High-Performance Sparse Communication for Machine Learning. *CoRR* abs/1802.08021 (2018). arXiv:1802.08021 http://arxiv.org/abs/1802.08021

Christopher De Sa, Ce Zhang, Kunle Olukotun, and Christopher Ré. 2015. Taming the Wild: A Unified Analysis of HOG WILD! -style Algorithms. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'15)*. MIT Press, Cambridge, MA, USA, 2674–2682.

Tim Salimans and Diederik P Kingma. 2016. Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 901–909. http://papers.nips.cc/paper/6114-weight-normalization-a-simple-reparameterization-to-accelerate-training-of-deep-neural-networks.pdf

JÃijrgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (2015), 85 – 117. https://doi.org/10.1016/j.neunet.2014.09.003

Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014a. 1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs, In Interspeech 2014. https://www.microsoft.com/en-us/research/publication/1-bit-stochastic-gradient-descent-and-application-to-data-parallel-distributed-training-of-speech-dnns/

F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 2014b. On parallelizability of stochastic gradient descent for speech DNNS. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 235–239. https://doi.org/10.1109/ICASSP.2014.6853593

Shai Shalev-Shwartz and Shai Ben-David. 2014. *Understanding machine learning: From theory to algorithms*. Cambridge university press.

Reza Shokri and Vitaly Shmatikov. 2015. Privacy-Preserving Deep Learning. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1310–1321. https://doi.org/10.1145/2810103.2813687

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354.

Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. *International Conference on Learning Representations (ICLR)* (2015). http://arxiv.org/abs/1409.1556

Andrew J. R. Simpson. 2015. Instant Learning: Parallel Deep Neural Networks and Convolutional Bootstrapping. *CoRR* abs/1505.05972 (2015). arXiv:1505.05972 http://arxiv.org/abs/1505.05972

Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. 2017. Don't Decay the Learning Rate, Increase the Batch Size. *CoRR* abs/1711.00489 (2017). arXiv:1711.00489 http://arxiv.org/abs/1711.00489

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2951–2959. http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf

Edgar Solomonik and Torsten Hoefler. 2015. Sparse Tensor Algebra as a Parallel Programming Model. (2015). arXiv:arXiv:1512.00066

M. Song, Y. Hu, H. Chen, and T. Li. 2017. Towards Pervasive and User Satisfactory CNN across GPU Microarchitectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1–12. https://doi.org/10.1109/HPCA.2017.52

H. V. Sorensen and C. S. Burrus. 1993. Efficient computation of the DFT with only a subset of input or output points. *IEEE Transactions on Signal Processing* 41, 3 (Mar 1993), 1184–1200. https://doi.org/10.1109/78.205723

Volker Strassen. 1969. Gaussian Elimination is Not Optimal. *Numer. Math.* 13, 4 (Aug. 1969), 354–356. https://doi.org/10.1007/BF02165411

Nikko Strom. 2015. Scalable distributed DNN training using commodity GPU cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*.

V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (Dec 2017), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Computer Vision and Pattern Recognition (CVPR)*. http://arxiv.org/abs/1409.4842

Gavin Taylor, Ryan Burmeister, Zheng Xu, Bharat Singh, Ankit Patel, and Tom Goldstein. 2016. Training Neural Networks Without Gradients: A Scalable ADMM Approach. *CoRR* abs/1605.02026 (2016). arXiv:1605.02026 http://arxiv.org/abs/1605.02026

A. N. Tikhonov. 1995. *Numerical methods for the solution of ill-posed problems*. Vol. 328. Springer.

J. Tsitsiklis, D. Bertsekas, and M. Athans. 1986. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Trans. Automat. Control* 31, 9 (Sep 1986), 803–812. https://doi.org/10.1109/TAC.1986.1104412

Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. 2011. Improving the speed of neural networks on CPUs. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*.

Nicolas Vasilache, Jeff Johnson, Michaël Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2015. Fast Convolutional Nets With fbfft: A GPU Performance Evaluation. *International Conference on Learning Representations (ICLR)* (2015). http://arxiv.org/abs/1412.7580

N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

P. Verbancsics and J. Harguess. 2015. Image Classification Using Generative Neuro Evolution for Deep Learning. In *2015 IEEE Winter Conference on Applications of Computer Vision*. 488–493. https://doi.org/10.1109/WACV.2015.71

André Viebke, Suejb Memeti, Sabri Pllana, and Ajith Abraham. 2017. CHAOS: a parallelization scheme for training convolutional neural networks on Intel Xeon Phi. *The Journal of Supercomputing* (06 Mar 2017). https://doi.org/10.1007/s11227-017-1994-x

Haohan Wang, Bhiksha Raj, and Eric P. Xing. 2017. On the Origin of Deep Learning. *CoRR* abs/1702.07800 (2017). arXiv:1702.07800 http://arxiv.org/abs/1702.07800

Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 1509–1519. http://papers.nips.cc/paper/6749-terngrad-ternary-gradients-to-reduce-communication-in-distributed-deep-learning.pdf

P. J. Werbos. 1990. Backpropagation through time: what it does and how to do it. *Proc. IEEE* 78, 10 (Oct 1990), 1550–1560. https://doi.org/10.1109/5.58337

Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8, 3 (01 May 1992), 229–256. https://doi.org/10.1007/BF00992696

Shmuel Winograd. 1980. *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104).

L. Xie and A. Yuille. 2017. Genetic CNN. In *2017 IEEE International Conference on Computer Vision (ICCV)*, Vol. 00. 1388–1397. https://doi.org/10.1109/ICCV.2017.154

Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric Xing. 2016. Lighter-communication Distributed Machine Learning via Sufficient Factor Broadcasting. In *Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence (UAI'16)*. AUAI Press, Arlington, Virginia, United States, 795–804. http://dl.acm.org/citation.cfm?id=3020948.3021030

E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. 2015. Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE Transactions on Big Data* 1, 2 (June 2015), 49–67. https://doi.org/10.1109/TBDATA.2015.2472014

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. 2015. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. *CoRR* abs/1502.03044 (2015). arXiv:1502.03044 http://arxiv.org/abs/1502.03044

Omry Yadan, Keith Adams, Yaniv Taigman, and Marc'Aurelio Ranzato. 2013. Multi-GPU Training of ConvNets. *CoRR* abs/1312.5853 (2013). http://arxiv.org/abs/1312.5853

Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. 2015. Performance Modeling and Scalability Optimization of Distributed Deep Learning Systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*. ACM, New York, NY, USA, 1355–1364. https://doi.org/10.1145/2783258.2783270

Yang You, Aydin Buluç, and James Demmel. 2017a. Scaling Deep Learning on GPU and Knights Landing Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 9, 12 pages. https://doi.org/10.1145/3126908.3126912

Yang You, Igor Gitman, and Boris Ginsburg. 2017b. Large Batch Training of Convolutional Networks. *CoRR* abs/1708.03888 (2017). arXiv:1708.03888 http://arxiv.org/abs/1708.03888

Yang You, Zhao Zhang, Cho-Jui Hsieh, and James Demmel. 2017c. 100-epoch ImageNet Training with AlexNet in 24 Minutes. *CoRR* abs/1709.05011 (2017). arXiv:1709.05011 http://arxiv.org/abs/1709.05011

Steven R. Young, Derek C. Rose, Travis Johnston, William T. Heller, Thomas P. Karnowski, Thomas E. Potok, Robert M. Patton, Gabriel Perdue, and Jonathan Miller. 2017. Evolving Deep Networks Using HPC. In *Proceedings of the Machine Learning on HPC Environments (MLHPC'17)*. ACM, New York, NY, USA, Article 7, 7 pages. https://doi.org/10.1145/3146347.3146355

Fisher Yu and Vladlen Koltun. 2016. Multi-Scale Context Aggregation by Dilated Convolutions. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1511.07122

Y. Yu, J. Jiang, and X. Chi. 2016. Using Supercomputer to Speed up Neural Network Training. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. 942–947. https://doi.org/10.1109/ICPADS.2016.0126

Hao Zhang, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, and Eric P. Xing. 2015. Poseidon: A System Architecture for Efficient GPU-based Deep Learning on Multiple Machines. *CoRR* abs/1512.06216 (2015). arXiv:1512.06216 http://arxiv.org/abs/1512.06216

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. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 181–193. https://www.usenix.org/conference/atc17/technical-sessions/presentation/zhang

Jian Zhang, Ioannis Mitliagkas, and Christopher Ré. 2017. YellowFin and the Art of Momentum Tuning. *CoRR* abs/1706.03471 (2017). arXiv:1706.03471 http://arxiv.org/abs/1706.03471

K. Zhang and X. W. Chen. 2014. Large-Scale Deep Belief Nets With MapReduce. *IEEE Access* 2 (2014), 395–403. https://doi.org/10.1109/ACCESS.2014.2319813

Sixin Zhang, Anna Choromanska, and Yann LeCun. 2015. Deep Learning with Elastic Averaging SGD. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'15)*. MIT Press, Cambridge, MA, USA, 685–693. http://dl.acm.org/citation.cfm?id=2969239.2969316

S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. 2016a. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783723

S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu. 2013. Asynchronous stochastic gradient descent for DNN training. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 6660–6663. https://doi.org/10.1109/ICASSP.2013.6638950

Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. 2016b. Staleness-aware async-SGD for Distributed Deep Learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*. AAAI Press, 2350–2356. http://dl.acm.org/citation.cfm?id=3060832.3060950

Xiru Zhang, Michael McKenna, Jill P. Mesirov, and David L. Waltz. 1990. An Efficient Implementation of the Back-propagation Algorithm on the Connection Machine CM-2. In *Advances in Neural Information Processing Systems 2*, D. S. Touretzky (Ed.). Morgan-Kaufmann, 801–809. http://papers.nips.cc/paper/281-an-efficient-implementation-of-the-back-propagation-algorithm-on-the-connection-machine-cm-2.pdf

H. Zhao and J. Canny. 2014. Kylix: A Sparse Allreduce for Commodity Clusters. In *2014 43rd International Conference on Parallel Processing*. 273–282. https://doi.org/10.1109/ICPP.2014.36

Zhao Zhong, Junjie Yan, and Cheng-Lin Liu. 2017. Practical Network Blocks Design with Q-Learning. *CoRR* abs/1708.05552 (2017). arXiv:1708.05552 http://arxiv.org/abs/1708.05552

Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. 2016. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *CoRR* abs/1606.06160 (2016). arXiv:1606.06160 http://arxiv.org/abs/1606.06160

Martin A. Zinkevich, Markus Weimer, Alex Smola, and Lihong Li. 2010. Parallelized Stochastic Gradient Descent. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2 (NIPS'10)*. Curran Associates Inc., USA, 2595–2603.

A. Zlateski, K. Lee, and H. S. Seung. 2016. ZNNi: Maximizing the Inference Throughput of 3D Convolutional Networks on CPUs and GPUs. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. 854–865. https://doi.org/10.1109/SC.2016.72

Barret Zoph and Quoc V. Le. 2017. Neural Architecture Search with Reinforcement Learning. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1611.01578

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2017. Learning Transferable Architectures for Scalable Image Recognition. *CoRR* abs/1707.07012 (2017). arXiv:1707.07012 http://arxiv.org/abs/1707.07012

A DNN LAYER COMPUTATION FORMULAS

A.1 Preamble

The loss function is ℓ, we are computing the tensor y from the input tensor x, using a layer function f with parameters w. Overall, the following three functions have to be computed:

(1) **Forward evaluation**: $y \equiv f(w, x)$
(2) **Gradient w.r.t. parameters**: $\nabla w \equiv \frac{\partial \ell}{\partial w} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial w}$ (chain rule).
(3) **Gradient backpropagation**: $\nabla x \equiv \frac{\partial \ell}{\partial x} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial x}$.

In the backpropagation algorithm, we use the computed ∇x to compute the gradients of the preceding layers in the DNN.

A.2 Activation

- $x \in \mathbb{R}^{W\times H\times C\times N}$; No parameters, thus w is nonexistent;
- $f(x) = \sigma(x)$;
- $\nabla x_{i,j,k,l} = \frac{\partial \ell}{\partial y_{i,j,k,l}} \cdot \sigma'(x_{i,j,k,l})$.
- Examples of activation functions:

(1) $ReLU(x) = \max\{0, x\}$; $ReLU'(x) = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases}$.
(2) Sigmoidal function: $\sigma(x) = \frac{1}{1+e^{-x}}$; $\sigma'(x) = \sigma(x)\cdot(1-\sigma(x))$.

A.3 Fully Connected Layers

- $w \in \mathbb{R}^{C_{out}\times C_{in}}$, $x \in \mathbb{R}^{C_{in}\times N}$, and $\frac{\partial \ell}{\partial y} \in \mathbb{R}^{C_{out}\times N}$;
- $f(w, x) = w \cdot x$;
- $\nabla w = x \cdot \left(\frac{\partial \ell}{\partial y}\right)^T$;
- $\nabla x = w \cdot \frac{\partial \ell}{\partial y}$.

A.4 Convolution (Direct)

- $w \in \mathbb{R}^{C_{out}\times C_{in}\times K_y\times K_x}$, $x \in \mathbb{R}^{W\times H\times C_{in}\times N}$, $y \in \mathbb{R}^{W'\times H'\times C_{out}\times N}$, and $\frac{\partial \ell}{\partial y} \in \mathbb{R}^{W'\times H'\times C_{out}\times N}$;
- W', H' are the output sizes of the convolution, defined as $W' = \left\lfloor \frac{W-K_x+2Pad_x}{Stride_x} \right\rfloor + 1$ and $H' = \left\lfloor \frac{H-K_y+2Pad_y}{Stride_y} \right\rfloor + 1$.
- Using input channel c and output channel c':

(1) $y_{c'} = \sum_{c=0}^{C_{in}} x_c * w_{c',c}$;
(2) $\frac{\partial \ell}{\partial x_c} = \frac{\partial \ell}{\partial y_{c'}} * w_{c',c}^T$;
(3) $\frac{\partial \ell}{\partial w_{c',c}} = \frac{\partial \ell}{\partial y_{c'}} * x_c$.

A.5 Pooling

- No parameters, $x, y, \frac{\partial \ell}{y} \in \mathbb{R}^{W\times H\times C\times N}$;
- $y_{i,j,k,l} = \max_{k_x\in[-K_x,K_x], k_y\in[-K_y,K_y]} x_{i+k_x, j+k_y, k, l}$;
- $\nabla x_{i,j,k,l} = \begin{cases} 1 & y_{i,j,k,l} = x_{i,j,k,l} \\ 0 & \text{otherwise} \end{cases}$.

A.6 Batch Normalization

- $w = \{\gamma_k, \beta_k\}_{k=0}^{C} \in \mathbb{R}^{2\times C}$, $x, y \in \mathbb{R}^{C\times N}$;

- Forward evaluation algorithm:

(1) $E_k \leftarrow \frac{1}{N}\sum_{i=0}^{N} x_{k,i}$;
(2) $V_k \leftarrow \frac{1}{N}\sum_{i=0}^{N}\left(x_{k,i} - E_k\right)^2$;
(3) $\hat{x}_{k,i} \leftarrow \frac{x_{k,i}-E_k}{\sqrt{V_k+\varepsilon}}$
(4) $y_{k,i} \leftarrow \gamma \cdot \hat{x}_{k,i} + \beta$

- Gradients:

(1) $\frac{\partial \ell}{\partial \gamma_k} = \sum_{i=0}^{N} \frac{\partial \ell}{\partial y_{k,i}} \hat{x}_{k,i}$
(2) $\frac{\partial \ell}{\partial \beta_k} = \sum_{i=0}^{N} \frac{\partial \ell}{\partial y_{k,i}}$

- Backpropagation (unsimplified):

(1) $\nabla\sigma_k = \sum_{m=0}^{N}\left(\frac{\partial \ell}{\partial y_{k,m}}\gamma_k \cdot (x_{k,m} - E_k) \cdot (V_k+\varepsilon)^{-3/2}\right)$
(2) $\frac{\partial \ell}{\partial x_{k,i}} = \frac{\partial \ell}{\partial y_{k,i}} \cdot \frac{\gamma_k}{\sqrt{V_k+\varepsilon}} - \frac{x_{k,i}-E_k}{N}\nabla\sigma_k + \frac{1}{N}\sum_{m=0}^{N}\left(\frac{\partial \ell}{\partial y_{k,m}} \cdot \frac{-\gamma_k}{\sqrt{V_k+\varepsilon}}\right) + \frac{\sum_{m=0}^{N}(x_{k,m}-E_k)}{N^2}\nabla\sigma_k$

B CONVOLUTION COMPUTATION ANALYSIS

Assuming convolution of a 4D tensor with a 4D kernel:

- Input tensor (x) shape: $N \times C_{in} \times H \times W$.
- Kernel tensor (w) shape: $C_{out} \times C_{in} \times K_y \times K_x$.
- Output tensor (y) shape: $N \times C_{out} \times H' \times W'$.
- In the general case $W' = \left\lfloor \frac{W-K_x+2P_x}{S_x} \right\rfloor + 1$ and $H' = \left\lfloor \frac{H-K_y+2P_y}{S_y} \right\rfloor + 1$ for padding P_x, P_y and strides S_x, S_y.
- However, assuming zero padding and a stride of 1 element we obtain $W' = W - K_x + 1$ and $H' = H - K_y + 1$.

B.1 Direct Convolution

Algorithm:

Algorithm 3 Direct Convolution

```
1:  for i = 0 to N in parallel do
2:      for j = 0 to C_out in parallel do
3:          for k = 0 to H' in parallel do
4:              for l = 0 to W' in parallel do
5:                  for m = 0 to C_in do                          ▷ Depth: log2 C_in
6:                      for k_y = 0 to K_y do                     ▷ Depth: log2 K_y
7:                          for k_x = 0 to K_x do                 ▷ Depth: log2 K_x
8:                              y_{i,j,k,l} += x_{i,m,k+k_y,l+k_x} · w_{j,m,k_y,k_x}   ▷ Work: 1
9:                          end for
10:                     end for
11:                 end for
12:             end for
13:         end for
14:     end for
15: end for
```

Overall cost:

$$\mathbf{W} = N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x$$
$$\mathbf{D} = \lceil \log_2 C_{in} \rceil + \lceil \log_2 K_y \rceil + \lceil \log_2 K_x \rceil$$

B.2 im2col

- Input matrix A is a result of a data-layout transformation, sized $\left(C_{in} \cdot K_y \cdot K_x\right) \times \left(N \cdot H' \cdot W'\right)$.
- Kernel matrix F is the reshaped tensor w, with dimensions $C_{out} \times \left(C_{in} \cdot K_y \cdot K_x\right)$.
- Output matrix B has a size of $C_{out} \times (N \cdot H' \cdot W')$ and is reshaped to the output.

Algorithm:

Algorithm 4 im2col Convolution

for $i = 0$ **to** $C_{in} \cdot K_y \cdot K_x$ **in parallel do**
  for $j = 0$ **to** $N \cdot H' \cdot W'$ **in parallel do**
    $A_{i,j} \leftarrow x_{\ldots}$ ▷ im2col. Work: 0, Depth: 0 (layout only)
  end for
end for

▷ Matrix Multiplication
$B \leftarrow F \cdot A$ ▷ Work: $C_{out} \cdot (C_{in} \cdot K_y \cdot K_x) \cdot (N \cdot H' \cdot W')$
▷ Depth: $\log_2\left(C_{in} \cdot K_y \cdot K_x\right)$

for $i = 0$ **to** N **in parallel do**
  for $j = 0$ **to** C_{out} **in parallel do**
    for $k = 0$ **to** H' **in parallel do**
      for $l = 0$ **to** W' **in parallel do**
        $y_{i,j,k,l} \leftarrow B_{\ldots}$ ▷ col2im. Work: 0, Depth: 0 (layout only)
      end for
    end for
  end for
end for

Overall cost:

$$\begin{aligned} \mathbf{W} &= N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x \\ \mathbf{D} &= \lceil \log_2 C_{in} \rceil + \lceil \log_2 K_y \rceil + \lceil \log_2 K_x \rceil \end{aligned}$$

B.3 FFT

- Formula: $y_{i,j,*,*} = \mathcal{F}^{-1}\left(\sum_{k=0}^{C_{in}} \mathcal{F}\left(x_{i,k,*,*}\right) \circ \mathcal{F}\left(w_{j,k,*,*}\right)\right)$, where $\circ$ denotes element-wise multiplication and w is padded to $H \cdot W$.
- $\hat{x}, \hat{w}$ are the transformed inputs and kernels reshaped as 3D tensors, obtained by batching 2-dimensional FFTs.
- We refer to $\hat{x}_k$ (or $\hat{w}_k$) as the kth 2D slice in the tensor.
- In practical implementations, $\hat{x}$ and $\hat{w}$ are reshaped to $W \cdot H$ 2D matrices (sized $N \times C_{in}$ and $C_{in} \times C_{out}$ respectively) to transform the point-wise multiplication and sum from the above formula to a batched complex matrix-matrix multiplication.

Algorithm:

Algorithm 5 FFT Convolution

for $i = 0$ **to** C_{out} **in parallel do**
  for $j = 0$ **to** C_{in} **in parallel do**
    $\hat{w}_{i,j} \leftarrow \mathcal{F}(w_{i,j})$ ▷ 2D FFT. Work: $c \cdot HW \log_2 HW$, Depth: $\log_2 HW$
  end for
end for
for $i = 0$ **to** N **in parallel do**
  for $j = 0$ **to** C_{in} **in parallel do**
    $\hat{x}_{i,j} \leftarrow \mathcal{F}(x_{i,j})$ ▷ 2D FFT. Work: $c \cdot HW \log_2 HW$, Depth: $\log_2 HW$
  end for
end for
for $i = 0$ **to** N **in parallel do**
  for $j = 0$ **to** C_{out} **in parallel do**
    for $k = 0$ **to** H' **in parallel do**
      for $l = 0$ **to** W' **in parallel do** ▷ Batched MM
        $\hat{y}_{i,j} = \sum_{m=0}^{C_{in}} \hat{x}_{i,m} \cdot \hat{w}_{j,m}$ ▷ Work: $H \cdot W \cdot N \cdot C_{in} \cdot C_{out}$
      end for ▷ Depth: $\log_2 C_{in}$
    end for
  end for
end for
for $i = 0$ **to** N **in parallel do**
  for $j = 0$ **to** C_{out} **in parallel do**
    $y_{i,j} \leftarrow \mathcal{F}^{-1}(\hat{y}_{i,j})$ ▷ 2D IFFT. Work: $c \cdot H'W' \log_2 H'W'$, Depth: $\log_2 H'W'$
  end for
end for

Overall cost:

$$\mathbf{W} = c \cdot HW \log_2(HW) \cdot (C_{out} \cdot C_{in} + N \cdot C_{in} + N \cdot C_{out}) + H \cdot W \cdot N \cdot C_{in} \cdot C_{out};$$

$$\mathbf{D} = 2\left\lceil \log_2 HW \right\rceil + \left\lceil \log_2 C_{in} \right\rceil;$$

where c is the hidden constant in 2D FFT.

B.4 Winograd

- Assuming $F(m \times m, r \times r)$ [Lavin and Gray 2016], i.e., output tile size of $m \times m$ and convolution kernel size is $r \times r$.
- $\alpha = m + r - 1$ is the input tile size. Neighboring tiles overlap by $r - 1$.
- Total number of tiles: $P = N \cdot \lceil H/m \rceil \cdot \lceil W/m \rceil$.
- $A \in \mathbb{R}^{\alpha \times m}, B \in \mathbb{R}^{\alpha \times \alpha}, G \in \mathbb{R}^{\alpha \times r}$ are Winograd minimal filtering matrices.

Algorithm:

Algorithm 6 Winograd Convolution

for $k = 0$ **to** C_{out} **in parallel do**
  for $c = 0$ **to** C_{in} **in parallel do**
    $u \leftarrow G w_{k,c,*,*} G^T$ ▷ Winograd transform $r \times r \rightarrow \alpha \times \alpha$. Work: $\mathbf{W}_{WG}$, Depth: $\mathbf{D}_{WG}$
    for $s = 0$ **to** α **in parallel do**
      for $n = 0$ **to** α **in parallel do**
        $U_{k,c}^{(s,n)} \leftarrow u_{s,n}$ ▷ Scatter. Work: 0, Depth: 0 (layout only)
      end for
    end for
  end for
end for
for $b = 0$ **to** P **in parallel do**
  for $c = 0$ **to** C_{in} **in parallel do**
    $v \leftarrow B^T x_{b,c,*,*} B$ ▷ Winograd transform $\alpha \times \alpha \rightarrow \alpha \times \alpha$. Work: $\mathbf{W}_{WD}$, Depth: $\mathbf{D}_{WD}$
    for $s = 0$ **to** α **in parallel do**
      for $n = 0$ **to** α **in parallel do**
        $V_{c,b}^{(s,n)} \leftarrow v_{s,n}$ ▷ Scatter. Work: 0, Depth: 0 (layout only)
      end for
    end for
  end for
end for
for $s = 0$ **to** α **in parallel do**
  for $n = 0$ **to** α **in parallel do**
    $Z^{(s,n)} \leftarrow U^{(s,n)} \cdot V^{(s,n)}$ ▷ Matrix Multiplication. Work: $C_{out} \cdot C_{in} \cdot P$, Depth: $\log_2 C_{in}$
  end for
end for
for $k = 0$ **to** C_{out} **in parallel do**
  for $b = 0$ **to** P **in parallel do**
    for $s = 0$ **to** α **in parallel do**
      for $n = 0$ **to** α **in parallel do**
        $z_{s,n} \leftarrow Z_{k,b}^{(s,n)}$ ▷ Gather. Work: 0, Depth: 0 (layout only)
      end for
    end for
    $y_{b,k,*,*} \leftarrow A^T z A$ ▷ Winograd transform $\alpha \times \alpha \rightarrow m \times m$. Work: $\mathbf{W}_{WY}$, Depth: $\mathbf{D}_{WY}$
  end for
end for

A Winograd transform from $a \times a \rightarrow b \times b$ consists of two matrix multiplications, thus:

$$\begin{aligned}\mathbf{W}_{WT}(a,b) &= b \cdot a \cdot a + b \cdot a \cdot b = a^2 b + b^2 a \\ \mathbf{D}_{WT}(a,b) &= 2\left\lceil \log_2 a \right\rceil;\end{aligned}$$

and therefore $\mathbf{W}_{WG} = r^2\alpha + \alpha^2 r, \mathbf{W}_{WD} = 2\alpha^3, \mathbf{W}_{WY} = \alpha^2 m + m^2\alpha$, and $\mathbf{D}_{WG} = 2\left\lceil \log_2 r \right\rceil, \mathbf{D}_{WD} = 2\left\lceil \log_2 \alpha \right\rceil, \mathbf{D}_{WY} = 2\left\lceil \log_2 \alpha \right\rceil$.

Overall cost:

$$\begin{aligned}\mathbf{W} &= \mathbf{W}_{WG} + \mathbf{W}_{WD} + C_{out} \cdot C_{in} \cdot P + \mathbf{W}_{WY} \\ &= \alpha(r^2 + \alpha r + 2\alpha^2 + \alpha m + m^2) + C_{out} \cdot C_{in} \cdot P \\ \mathbf{D} &= 2\left\lceil \log_2 r \right\rceil + 4\left\lceil \log_2 \alpha \right\rceil + \left\lceil \log_2 C_{in} \right\rceil\end{aligned}$$