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:

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

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. 1. Deep Learning Usage Examples (figures adapted from respective citations)
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>
<thead>
<tr>
<th>Year</th>
<th>2012</th>
<th>2013</th>
<th>2014</th>
<th>2015</th>
<th>2016</th>
<th>2017</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs.AI</td>
<td>1,081</td>
<td>1,765</td>
<td>1,022</td>
<td>1,105</td>
<td>1,929</td>
<td>2,790</td>
</tr>
<tr>
<td>cs.CV</td>
<td>577</td>
<td>852</td>
<td>1,349</td>
<td>2,261</td>
<td>3,627</td>
<td>5,693</td>
</tr>
</tbody>
</table>

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/
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 \( f : X \rightarrow Y \), we wish to minimize the loss function \( L_{\mathcal{D}}(f) = \mathbb{P}[f(z) \neq h(z)] \) (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 \( \theta \)), 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:

\[
\begin{align*}
w^* & = \arg \min_{w \in \mathcal{H}} L_{\mathcal{D}}(f_w) = \arg \min_{w \in \mathcal{H}} \mathbb{E}_{z \sim \mathcal{D}}[\ell(w, z)],
\end{align*}
\]

where \( \ell : \mathcal{H} \times X \rightarrow \mathbb{R}_+ \) 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 \( \ell \) 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: $\ell(w, z) = -\sum_i h(z)_i \log \sigma(f_w(z))_i$.

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 $D$ is never observed, it is necessary to obtain an unbiased estimator of the gradient. Observe that $\nabla L_D (w) = \mathbb{E}_{z \sim D} \left[ \nabla \ell (w, z) \right]$ (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 $D$. 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:

\begin{algorithm}
1: for $t = 0$ to $T$ do
2: $z \leftarrow \text{Random element from } S$ \hspace{1cm} \triangleright Stopping condition
3: $g \leftarrow \nabla \ell (w^{(t)}, z)$ \hspace{1cm} \triangleright Sample dataset $S$
4: $w^{(t+1)} \leftarrow w^{(t)} + u(g, w^{(0,\ldots,t)}, t)$ \hspace{1cm} \triangleright Compute gradient of $\ell$
5: end for
\end{algorithm}

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 $(w^{(t)})$ using backpropagation (see Section 3.2). The obtained gradient is then used for updating the current weights, using a weight update rule (line 4).
Two other classes in machine learning are reinforcement learning and unsupervised learning. 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].

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, the basic SGD update rule is $u_{sgd}(g) = -\eta \cdot g$, where $\eta$ represents the learning rate. $\eta$ controls how much the gradient values will overall affect the next estimate $w^{(t-1)}$, and in iterative nonlinear optimization methods finding the correct $\eta$ is a considerable part of the computation [Nocedal and Wright 2006]. In machine learning problems, it is customary to fix $\eta$, or set an iteration-based weight update rule $u_{alg}(g, t) = -\eta_t \cdot g$, where $\eta_t$ 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.

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 $w^{(0)}, \ldots, w^{(t)}$, and the current iteration $t$. Table 3 summarizes the popular $u$ functions used in training. In the table, the basic SGD update rule $u_{alg}(g, t) = -\eta_t \cdot g$, where $\eta_t$ decreases (decays) over time to bound the modification size and avoid local divergence.

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, $\mu, \beta, \beta_1, \text{ and } \beta_2$ 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

**Table 3. Popular Weight Update Rules**

<table>
<thead>
<tr>
<th>Method</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning Rate</td>
<td>$w^{(t+1)} = w^{(t)} - \eta \cdot \nabla f(w^{(t)}, z)$</td>
</tr>
<tr>
<td>Adaptive Learning Rate</td>
<td>$w^{(t+1)} = w^{(t)} - \eta_t \cdot \nabla f(w^{(t)})$</td>
</tr>
<tr>
<td>Momentum [Qian 1999]</td>
<td>$w^{(t+1)} = w^{(t)} + \mu \cdot (w^{(t)} - w^{(t-1)}) - \eta \cdot \nabla f(w^{(t)})$</td>
</tr>
<tr>
<td>Nesterov Momentum [Nesterov 1983]</td>
<td>$w^{(t+1)} = w^{(t)} + v_t$; $v_{t+1} = \mu \cdot v_t - \eta \cdot \nabla f(w^{(t)} - \mu \cdot v_t, z)$</td>
</tr>
<tr>
<td>AdaGrad [Duchi et al. 2011]</td>
<td>$w^{(t+1)} = w^{(t)} - \frac{\eta \cdot \nabla f(w^{(t)})}{\sqrt{A_{t-1} + \epsilon}}$</td>
</tr>
<tr>
<td>RMSProp [Hinton 2012]</td>
<td>$w^{(t+1)} = w^{(t)} - \frac{\eta \cdot \nabla f(w^{(t)})}{\sqrt{\beta_2 A_{t-1} + \epsilon}}$</td>
</tr>
<tr>
<td>Adam [Kingma and Ba 2015]</td>
<td>$w^{(t+1)} = w^{(t)} - \frac{\eta \cdot M_{t-1}^{(m)}}{\sqrt{\beta_2 M_{t-1}^{(m)}}}$</td>
</tr>
</tbody>
</table>

where $f$ represents the objective function, $\nabla f$ its gradient with respect to $w$, $\eta$ the learning rate, and $A_t, r_t, v_t, M_t$ momentum, RMS decay rate, and first and second moment decay rate. In the table, $\mu, \beta, \beta_1, \text{ and } \beta_2$ represent the momentum, RMS decay rate, and first and second moment decay rate hyper-parameters, respectively.
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 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 $W$, which corresponds to the total number of vertices and the graph’s depth $D$, 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 $T_1 = W$ and the time needed to process the graph on an infinite number of processes is $T_\infty = D$. The average parallelism in the computation is $W/D$, 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: $\min\{W/p, D\} \leq T_p \leq O(W/p + D)$ [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 $W = n - 1$ and $D = \log_2 n$, 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 ($\sigma = g = 0$), which makes it similar to the simple $\alpha$-$\beta$ 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 $T_r \geq \gamma mG$, where $\gamma$ represents the
size of a single data value and \( m \) is the number of values sent. This bound can be strengthened to 
\[ T_r \geq L \log_2(P) + 2ymG(P - 1)/P \] 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 
\[ T_{tree} = 2 \log_2(P)(L + ymG). \]
Yet, this algorithm is inefficient and can be optimized with a simple butterfly pattern, reducing the time to 
\[ T_{bfly} = \log_2(P)(L + ymG). \]
The butterfly algorithm is efficient (near-optimal) for small \( \gamma m \). For large \( \gamma m \) 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 \( P \): 
\[ T_{pipe} = 2(P-1)(L + \frac{ym}{P}G). \]
For most ranges of \( \gamma m \) and \( P \), one could use Rabenseifner’s algorithm [Rabenseifner 2004], which combines reduce-scatter with gather, running in time 
\[ T_{rabe} = 2L \log_2(P) + 2ymG(P - 1)/P. \]
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

1: \textbf{for} \( t = 0 \) to \( \frac{|S|}{B} \cdot \text{epochs} \) \textbf{do}
2: \hspace{1em} \( \tilde{z} \leftarrow \text{Sample } B \text{ elements from } S \) \hspace{1em} \( \triangleright \) Obtain samples from dataset
3: \hspace{1em} \( w_{mb} \leftarrow w^{(t)} \) \hspace{1em} \( \triangleright \) Load parameters
4: \hspace{1em} \( f \leftarrow \ell(w_{mb}, \tilde{z}, h(\tilde{z})) \) \hspace{1em} \( \triangleright \) Compute forward evaluation
5: \hspace{1em} \( g_{mb} \leftarrow \nabla \ell(w_{mb}, f) \) \hspace{1em} \( \triangleright \) Compute gradient using backpropagation
6: \hspace{1em} \( \Delta w \leftarrow u(g_{mb}, w^{(0, \ldots, t)}, t) \) \hspace{1em} \( \triangleright \) Weight update rule
7: \hspace{1em} \( w^{(t+1)} \leftarrow w_{mb} + \Delta w \) \hspace{1em} \( \triangleright \) Store parameters
8: \textbf{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.

![Minibatch Effect on Accuracy and Performance (Illustration)](image1)

![Empirical Accuracy (ResNet-50, figure adapted from Goyal et al. 2017, lower is better)](image2)

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.
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 $\sigma$) 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:

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

(2)
### Name Description
| N  | Minibatch size |
| C  | Number of channels/neurons |
| H  | Image Height |
| W  | Image Width |
| Kx | Convolution kernel width |
| Ky | Convolution kernel height |

(a) Data Dimensions

![Data Dimensions](image)

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:

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

where $\gamma, \beta$ are scaling factors, and $\epsilon$ 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
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 — \( \nabla x \) (w.r.t. input data), and \( \nabla w_i \) (w.r.t. layer weights). Note that some layers do not maintain mutable parameters (e.g., pooling, concatenation), and thus \( \nabla w_i \) is not always computed.

Fig. 11. The Backpropagation Algorithm
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 \( \ell_1 \) or \( \ell_2 \) 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 \( \frac{\text{thres}}{\|V\|} \). 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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Publication</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>w</td>
<td></td>
<td>60K</td>
<td>61M</td>
<td>6.8M</td>
</tr>
<tr>
<td>Layers (∝ D)</td>
<td>7</td>
<td>13</td>
<td>27</td>
<td>50–152</td>
<td>40–250</td>
</tr>
<tr>
<td>Operations (∝ W)</td>
<td>N/A</td>
<td>725M</td>
<td>1566M</td>
<td>∼1000M–2300M</td>
<td>∼600M–1130M</td>
</tr>
<tr>
<td>(ImageNet-1k)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Top-5 Error</td>
<td>N/A</td>
<td>15.3%</td>
<td>9.15%</td>
<td>5.71%</td>
<td>5.29%</td>
</tr>
<tr>
<td>(ImageNet-1k)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Top-1 Error</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>6.41%</td>
<td>3.62%</td>
</tr>
<tr>
<td>(CIFAR-10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


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 ($W/D$) yields the following result (see Appendix A for details regarding each layer):

\[
W = W_{\text{conv}(32,5,1,6)} + W_{\text{pool}(28,2,6)} + W_{\text{conv}(14,5,6,16)} + W_{\text{pool}(10,2,16)} + W_{f_{c}(400,120)} + W_{f_{c}(120,84)} + W_{f_{c}(84,10)}
\]

\[
= 117,600 + 14,112 + 470,400 + 4,800 + 48,000 + 10,080 + 840
\]

\[
= 665,832
\]

\[
D = 5 + 2 + 9 + 2 + 9 + 7 + 7 = 41
\]

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

---

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.

---

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 $1 \times 1$ 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.
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.

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 $\sim 2$ 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...
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
As such, the research community and the industry have invested considerable efforts into optimizing (possibly with overlap), generating redundant information (see Fig. 18a). The convolution kernels (colloquially known as im2col) are geared towards many parallel multiplication-accumulation operations. It is, however, possible to increase the utilization by introducing data redundancy or basis transformation.

Table 6 summarizes their work and depth characteristics (see Appendix B for detailed analyses).

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%.

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%.
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:

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

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 $\leq 32$ 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:

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

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
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 ($W/D$) 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 $27.9 \times$ performance increase over the state-of-the-art for a single layer, and $5.6 \times$ 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
next layer once its data dependencies have been met. Overall, these optimizations result in $\sim11\times$ 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 $\sim33\%$ 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 $\sim30\times$ 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 ($W/D$) 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 ($\propto W$) of the layers, and the depth of the DNN ($\propto D$), 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

<table>
<thead>
<tr>
<th>Category</th>
<th>Papers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Parallelism</td>
<td>[Coates et al. 2013; Ericson and Mbuva 2017; Lee et al. 2015; Muller and Gunzinger 1994; Ngiam et al. 2010; Yadan et al. 2013]</td>
</tr>
<tr>
<td>Layer Pipelining</td>
<td>[Chilimbi et al. 2014; Dean et al. 2012; Deng et al. 2012]</td>
</tr>
<tr>
<td>Hybrid Parallelism</td>
<td>[Ben-Nun et al. 2015; Chilimbi et al. 2014; Dean et al. 2012; Gaunt et al. 2017; Krizhevsky 2014; Viebke et al. 2017]</td>
</tr>
</tbody>
</table>

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 (D) of the operator from \( O(\log N) \) to \( O(1) \), removing inter-dependencies within the minibatch. According to the authors,
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
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
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 \(~2.2\times\) [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 \(~6.25\times\) can be achieved for 8 GPUs over one,
with less than 1% accuracy loss. These results were also reaffirmed in other hybrid implementations [Ben-Nun et al. 2015], in which $3.1 \times$ 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 $3.94 \times$ 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

<table>
<thead>
<tr>
<th>Category</th>
<th>Method</th>
<th>Papers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model Consistency</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronization</td>
<td>Synchronous</td>
<td>[Coates et al. 2013; Iandola et al. 2016a; Moritz et al. 2016; Seide et al. 2014a; Strom 2015; Yadan et al. 2015]</td>
</tr>
<tr>
<td>Asynchronous</td>
<td></td>
<td>[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]</td>
</tr>
<tr>
<td>Nondeterministic Comm.</td>
<td></td>
<td>[Jin et al. 2016; Ram et al. 2009]</td>
</tr>
<tr>
<td><strong>Parameter Distribution and Communication</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Centralization</td>
<td>Parameter Server (PS)</td>
<td>[Cui et al. 2016; Iandola et al. 2016a; Kim et al. 2016a; Li et al. 2014]</td>
</tr>
<tr>
<td>Decentralized</td>
<td></td>
<td>[Jin et al. 2016; Lian et al. 2017]</td>
</tr>
<tr>
<td>Sparsification</td>
<td></td>
<td>[Aji and Heafield 2017; Chen et al. 2017; Dryden et al. 2016; Lin et al. 2018; Ng et al. 2018; Shokri and Shmatikov 2015; Strom 2015]</td>
</tr>
<tr>
<td><strong>Distributed Training</strong></td>
<td>Ensemble Learning</td>
<td>[Im et al. 2016; Lee et al. 2015; Simpson 2015]</td>
</tr>
<tr>
<td>Model Consolidation</td>
<td>Knowledge Distillation</td>
<td>[Ba and Caruana 2014; Hinton et al. 2015]</td>
</tr>
<tr>
<td>Model Averaging</td>
<td>Direct</td>
<td>[Blanchard et al. 2017; Chen and Huo 2016; Miao et al. 2014; Zinkevich et al. 2010]</td>
</tr>
<tr>
<td></td>
<td>Elastic Averaging</td>
<td>[Jin et al. 2016; Lian et al. 2017; You et al. 2017a; Zhang et al. 2015]</td>
</tr>
<tr>
<td></td>
<td>Natural Gradient</td>
<td>[Povey et al. 2014]</td>
</tr>
</tbody>
</table>

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(τ,i) for τ ≤ t, where t − τ 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 fω, 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].
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 \( \nabla w \) 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...
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 $\nabla w$ 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
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 [Detmers 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 [Detmers 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
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 ($\propto D$) for bandwidth ($\propto W$), 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 $1 \times 1$ 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,
exploit (a) Sequential Optimisation (b) Parallel Random/Grid Search (c) Population Based Training

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;
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 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 4x 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


Dario Amodei, Sundaram Ananthanarayan, Rishita Anubhui, 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, Sherijl Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Qian, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya
Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis


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


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


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


A DNN LAYER COMPUTATION FORMULAS

A.1 Preamble

The loss function is $\ell$, 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 $\nabla x$ 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** ($x > 0$) $\sigma(x) = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise}. \end{cases}$

   \[ \sigma'(x) = \sigma(x) \cdot (1 - \sigma(x)). \]

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 $W' = \left\lfloor \frac{W - K_y + 2 \text{Pad}_y}{\text{Stride}_x} \right\rfloor + 1$ and $H' = \left\lfloor \frac{H - K_x + 2 \text{Pad}_x}{\text{Stride}_y} \right\rfloor + 1$.
- Using input channel $c$ and output channel $c'$:
  1. $y_{c'} = \sum_{c=0}^{C_{in}} x_c * w_{c', c}$;
  2. $\frac{\partial \ell}{\partial x_c} = \frac{\partial \ell}{\partial y_{c'}} * w_{c', c}$;
  3. $\frac{\partial \ell}{\partial y_{c'}} = \frac{\partial \ell}{\partial y_{c'}} * x_c$.

A.5 Pooling

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

A.6 Batch Normalization

- $w = \{y_k, \beta_k\}_{k=0}^C \in \mathbb{R}^{2 \times C}$, $x, y \in \mathbb{R}^{C \times N}$;
Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis

**Forward evaluation algorithm:**
1. \( E_k \leftarrow \frac{1}{N} \sum_{i=0}^{N} x_{k,i} \);
2. \( V_k \leftarrow \frac{1}{N} \sum_{i=0}^{N} (x_{k,i} - E_k)^2 \);
3. \( \hat{x}_{k,i} \leftarrow \frac{x_{k,i} - E_k}{\sqrt{V_k + \epsilon}} \);
4. \( y_{k,i} \leftarrow y \cdot \hat{x}_{k,i} + \beta \)

**Gradients:**
1. \( \frac{\partial \ell}{\partial y_k} = \sum_{i=0}^{N} \frac{\partial \ell}{\partial y_{k,i}} \cdot \hat{x}_{k,i} \)
2. \( \frac{\partial \ell}{\partial \beta_k} = \sum_{i=0}^{N} \frac{\partial \ell}{\partial y_{k,i}} \)

**Backpropagation (unsimplified):**
1. \( \nabla \sigma_k = \sum_{m=0}^{N} \left( \frac{\partial \ell}{\partial y_{k,m}} \gamma_k \cdot (x_{k,m} - E_k) \cdot (V_k + \epsilon)^{-3/2} \right) \)
2. \( \frac{\partial \ell}{\partial x_{k,i}} = \frac{\partial \ell}{\partial y_{k,i}} \cdot \frac{y_k}{\sqrt{V_k + \epsilon}} - \frac{x_{k,i} - E_k}{N} \nabla \sigma_k + \frac{1}{N} \sum_{m=0}^{N} \left( \frac{\partial \ell}{\partial y_{k,m}} \cdot \frac{-y_k}{\sqrt{V_k + \epsilon}} \right) + \frac{\sum_{m=0}^{N} (x_{k,m} - E_k)}{N} \nabla \sigma_k \)
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
  \[
  W' = \left\lfloor \frac{W - K_x + 2P_x}{S_x} \right\rfloor + 1 \quad \text{and} \quad H' = \left\lfloor \frac{H - K_y + 2P_y}{S_y} \right\rfloor + 1
  \]
  for padding \( P_x, P_y \) and strides \( S_x, S_y \).
- However, assuming zero padding and a stride of 1 element we obtain
  \[
  W' = W - K_x + 1 \quad \text{and} \quad H' = H - K_y + 1.
  \]

B.1 Direct Convolution

Algorithm:

\begin{algorithm}
\caption{Direct Convolution}
\begin{algorithmic}[1]
\For {i = 0 \textbf{to} N in parallel}
  \For {j = 0 \textbf{to} C_{out} in parallel}
    \For {k = 0 \textbf{to} H' in parallel}
      \For {l = 0 \textbf{to} W' in parallel}
        \For {m = 0 \textbf{to} C_{in}}\Comment{Depth: \( \log_2 C_{in} \)}
          \For {k_y = 0 \textbf{to} K_y}\Comment{Depth: \( \log_2 K_y \)}
            \For {k_x = 0 \textbf{to} K_x}\Comment{Depth: \( \log_2 K_x \)}
              \State \( y_{i,j,k,l} += x_{i,m,k+k_y,l+k_x} \cdot w_{j,m,k_y,k_x} \)\Comment{Work: 1}
            \EndFor
          \EndFor
        \EndFor
      \EndFor
    \EndFor
  \EndFor
\EndFor
\end{algorithmic}
\end{algorithm}

Overall cost:
\[
W = N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x
\]
\[
D = \left\lfloor \log_2 C_{in} \right\rfloor + \left\lfloor \log_2 K_y \right\rfloor + \left\lfloor \log_2 K_x \right\rfloor
\]
B.2 im2col

- Input matrix $A$ is a result of a data-layout transformation, sized $(C_{in} \cdot K_y \cdot K_x) \times (N \cdot H' \cdot W')$.
- Kernel matrix $F$ is the reshaped tensor $w$, with dimensions $C_{out} \times (C_{in} \cdot K_y \cdot K_x)$.
- 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**

```plaintext
1: for $i = 0$ to $C_{in} \cdot K_y \cdot K_x$ in parallel do
2:     for $j = 0$ to $N \cdot H' \cdot W'$ in parallel do
3:         $A_{i,j} \leftarrow x$...
4:     end for
5: end for
6: $B \leftarrow F \cdot A$  \hspace{1cm} \triangleright \text{Matrix Multiplication}
7: $B_{i,j} \leftarrow A_{i,j} \cdot F$...
8: for $i = 0$ to $N$ in parallel do
9:     for $j = 0$ to $C_{out}$ in parallel do
10:        for $k = 0$ to $H'$ in parallel do
11:            for $l = 0$ to $W'$ in parallel do
12:                $y_{i,j,k,l} \leftarrow B_{i,j,k,l}$...
13:            end for
14:        end for
15:    end for
16: end for
```

Overall cost:

- $W = N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x$
- $D = \left\lfloor \log_2 C_{in} \right\rfloor + \left\lfloor \log_2 K_y \right\rfloor + \left\lfloor \log_2 K_x \right\rfloor$
B.3 FFT

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

1: for $i = 0$ to $C_{out}$ in parallel do
2:   for $j = 0$ to $C_{in}$ in parallel do
3:     $\hat{w}_{i,j} \leftarrow \mathcal{F}(w_{i,j})$ \hspace{1cm} $\triangleright$ 2D FFT. Work: $c \cdot HW \log_2 HW$, Depth: $\log_2 HW$
4:   end for
5: end for
6: for $i = 0$ to $N$ in parallel do
7:   for $j = 0$ to $C_{in}$ in parallel do
8:     $\hat{x}_{i,j} \leftarrow \mathcal{F}(x_{i,j})$ \hspace{1cm} $\triangleright$ 2D FFT. Work: $c \cdot HW \log_2 HW$, Depth: $\log_2 HW$
9:   end for
10: end for
11: for $i = 0$ to $N$ in parallel do
12:   for $j = 0$ to $C_{out}$ in parallel do
13:     for $k = 0$ to $H'$ in parallel do
14:       for $l = 0$ to $W'$ in parallel do
15:         $\hat{y}_{i,j} = \sum_{m=0}^{C_{in}} \hat{x}_{i,m} \cdot \hat{w}_{j,m}$ \hspace{1cm} $\triangleright$ Work: $H \cdot W \cdot N \cdot C_{in} \cdot C_{out}$
16:       end for
17:     end for
18:   end for
19: end for
20: for $i = 0$ to $N$ in parallel do
21:   for $j = 0$ to $C_{out}$ in parallel do
22:     $y_{i,j} \leftarrow \mathcal{F}^{-1}(\hat{y}_{i,j})$ \hspace{1cm} $\triangleright$ 2D IFFT. Work: $c \cdot H'W' \log_2 H'W'$, Depth: $\log_2 H'W'$
23:   end for
24: end for

Overall cost:

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

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

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**

1: for $k = 0$ to $C_{out}$ in parallel do
2:     for $c = 0$ to $C_{in}$ in parallel do
3:         $u \leftarrow Gw_{k,c,s}G^T$ \text{ \text{\quad } Winograd\ transform\ } r \times r \rightarrow \alpha \times \alpha.\ \text{Work: } W_{W,G},\ \text{Depth: } D_{W,G}$
4:             for $s = 0$ to $\alpha$ in parallel do
5:                 for $n = 0$ to $\alpha$ in parallel do
6:                     $U_{k,c}^{(s,n)} \leftarrow u_{s,n} \quad \text{\quad } \text{Scatter. Work: 0, Depth: 0 (layout only)}$
7:             end for
8:         end for
9:     end for
10: end for
11: for $b = 0$ to $P$ in parallel do
12:     for $c = 0$ to $C_{in}$ in parallel do
13:         $v \leftarrow B^T x_{b,c,s}B$ \text{ \text{\quad } Winograd\ transform\ } \alpha \times \alpha \rightarrow \alpha \times \alpha.\ \text{Work: } W_{W,D},\ \text{Depth: } D_{W,D}$
14:             for $s = 0$ to $\alpha$ in parallel do
15:                 for $n = 0$ to $\alpha$ in parallel do
16:                     $V_{c,b}^{(s,n)} \leftarrow v_{s,n} \quad \text{\quad } \text{Scatter. Work: 0, Depth: 0 (layout only)}$
17:             end for
18:         end for
19:     end for
20: end for
21: for $s = 0$ to $\alpha$ in parallel do
22:     for $n = 0$ to $\alpha$ in parallel do
23:         $Z^{(s,n)} \leftarrow U^{(s,n)} \cdot V^{(s,n)} \quad \text{\quad } \text{Matrix\ Multiplication. Work: } C_{out} \cdot C_{in} \cdot P,\ \text{Depth: } \log_2 C_{in}$
24:     end for
25: end for
26: for $k = 0$ to $C_{out}$ in parallel do
27:     for $b = 0$ to $P$ in parallel do
28:         for $s = 0$ to $\alpha$ in parallel do
29:             for $n = 0$ to $\alpha$ in parallel do
30:                 $z_{s,n} \leftarrow Z^{(s,n)}_{k,b} \quad \text{\quad } \text{Gather. Work: 0, Depth: 0 (layout only)}$
31:             end for
32:         end for
33:     end for
34: end for
35: $y_{b,k,s} \leftarrow A^T zA \quad \text{\quad } \text{Winograd\ transform\ } \alpha \times \alpha \rightarrow m \times m.\ \text{Work: } W_{W,Y},\ \text{Depth: } D_{W,Y}$
36: end for
A Winograd transform from $a \times a \rightarrow b \times b$ consists of two matrix multiplications, thus:

\[
W_{WT}(a, b) = b \cdot a \cdot a + b \cdot a \cdot b = a^2 b + b^2 a
\]

\[
D_{WT}(a, b) = 2 \left\lceil \log_2 a \right\rceil
\]

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

Overall cost:

\[
W = W_{WG} + W_{WD} + C_{out} \cdot C_{in} \cdot P + W_{WY}
\]

\[
= \alpha (r^2 + \alpha r + 2\alpha^2 + \alpha m + m^2) + C_{out} \cdot C_{in} \cdot P
\]

\[
D = 2 \left\lceil \log_2 r \right\rceil + 4 \left\lceil \log_2 \alpha \right\rceil + \left\lceil \log_2 C_{in} \right\rceil
\]