Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

A Performance-Based Recommender System for Distributed DNN Training

Author: Matthijs Jansen (2655671, 11045663)

1st supervisor:	Dr. ir. A.L. Varbanescu	
daily supervisor:	Dr. ir. V. Codreanu	(SURFsara)
2nd reader:	Prof. dr. ir. A. Iosup	

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

September 17, 2020

"I know that two and two make four - and should be glad to prove it too if I could - though I must say if by any sort of process I could convert 2 and 2 into five it would give me much greater pleasure."

- George Gordon Byron, November 10 1813

Abstract

Due to its many applications across various fields of research, engineering, and daily life, deep learning has seen a surge in popularity. Therefore, larger and more expressive models have been proposed, with examples like Turing-NLG using as many as 17 billion parameters. Training these very large models becomes increasingly difficult due to the high computational costs and large memory footprint. Therefore, several distribution models for deep learning based on data parallelism (e.g., Horovod) and model/pipeline parallelism (e.g., GPipe, PipeDream) have emerged.

In this work, I present a recommender system that, given a model of a deep neural network, a dataset, and a hardware configuration, recommends the fastest distribution model to the user. To do so, I build a performance predictor that, using analytical and statistical methods, predicts the best-performing distribution model out of a list of state-of-the art candidates. A comprehensive distributed deep learning benchmark suite is created to test the predictor with.

Through in-depth performance analysis and experimentation with various models, datasets, distribution models and hardware systems, I demonstrate that the proposed benchmark suite can accurately evaluate the capability of a given system to perform distributed machine learning training. I show that data parallelism can achieve almost linear speedups when training compute intensive models like ResNet, while models like VGG suffer from increased communication requirements in large clusters. Furthermore, I show that pipeline parallelism is able to achieve speedups in computation times with an increasing number of workers where model parallelism can not, while still benefiting from reduced memory footprints.

The performance predictor predicts the training time of compute intensive tasks for Horovod with less than 10 percent deviation, while for communication intensive tasks the deviation may increase as the communication patterns between

workers are very dynamic in nature, making them hard to predict. With in-depth analysis I prove that both torchgpipe and PipeDream do not function as intended, showing the immaturity of pipeline parallelism as a new distributed algorithm. As a result, the current iteration of the recommender system may not always accurately predict the fastest distribution model as both torchgpipe and PipeDream implementations perform worse than their intended designs suggest.

Contents

List of Figures

LIST OF FIGURES

List of Tables

1

Introduction

Deep learning has seen a rapid increase in progress in recent years, with milestones such as AlphaGo beating the world champion Go in 2016 (2), and the emergence of deep fakes in 2019 (3). This growth can be attributed to three factors. Firstly, more complex and efficient deep neural networks (DNNs) have been created, which are used for diverse fields of research such as image classification (4) and natural language processing (5). Secondly, the field of deep learning has opened up to researchers without a deep learning background through the introduction of user-friendly deep learning frameworks such as TensorFlow (6), Keras (7) and PyTorch (8). Finally, larger models are trained on more data in a smaller amount of time because the compute power and memory capacity of hardware systems has increased (9).

These three factors are intertwined: an increase in compute power can only be leveraged if deep learning frameworks support neural networks and distributed algorithms which can make use of it. An example of this is (10): researchers from Facebook trained a ResNet-50 model on the ImageNet dataset (1.28 million images) in only 1 hour by combining simple machine learning algorithms and distributing the task over 256 GPUs. Just two years later this record has been improved to 74.7 seconds (11) using 2048 GPUs. This evolution shows that the capability of a distributed algorithm to make use of the hardware is at least as important as the number of workers used in achieving low training times.

However, as the number of distributed algorithms and distribution models (implementations of distributed algorithms) increase, choosing the best distribution model in terms of training time for a particular deep learning application is difficult. These distribution models are of such complexity that it is often not possible to determine which distribution model will perform best without extensive testing. Moreover, with the size-explosion of

neural networks, many deep learning application require specific distribution models which lessen the memory usage per worker.

1.1 Research Question and Approach

The goal of this thesis is to create a recommender system that, given a model of a deep neural network, a dataset, and a hardware configuration, recommends the fastest distribution model to the user. To do so, I aim to build a performance predictor that, using analytical and statistical methods, predicts the best-performing distribution model out of a list of state-of-the-art candidates (Figure 1.1). The recommender system reduces the burden on the user by automating the benchmarking of the distribution models, which would require time and expertise.

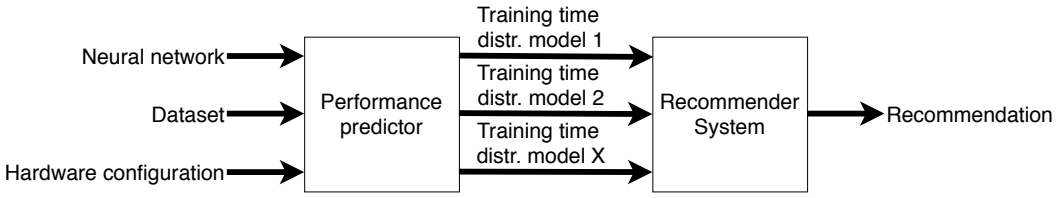


Figure 1.1: Design of the recommender system.

The predictor is tested using a distributed deep neural network benchmark suite specially created for this thesis. The current implementation includes, for datasets, MNIST (12), CIFAR-10 (13), ImageNet (14), and a high dimensional synthetic dataset; these datasets are included to support varying sizes and dimensions. Models-wise, I include MobileNetV2 (15), along with several ResNet (16) and VGG (17) models, thus supporting both compute- or communication-focused networks. Finally, as distribution models, I include Horovod (18) to represent data parallelism, along with GPipe (1) (as implemented in (19)) and PipeDream (20) for model and pipeline parallelism. PyTorch is used as framework as it is currently the only one which supports all three state-of-the-art distribution models.

I make the following contributions to the field of distributed deep learning:

I define a theoretical framework to compare distribution models for deep learning.

I present the design and implementation of a comprehensive benchmark suite for the evaluation of distributed DNNs.

I provide an in-depth empirical analysis, facilitated by my benchmark suite, on 3 distributed models, 6 networks, 4 datasets, and two different clusters.

I create a recommender system that can predict which distribution model is the best performing for a particular deep learning application.

1.2 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2 I introduce the distribution models and related work in the fields of distributed deep learning and performance prediction. In Chapter 3 the design and implementation of the benchmark suite is explained. Chapters 4 to 7 contain the performance analysis of and experiments on the predictors for respectively sequential deep learning, Horovod, GPipe and PipeDream. In Chapter 8 the accuracy of the performance predictor and recommender system is discussed, along with possible alternative implementations for the recommender system. Finally in Chapter 9 I look back on the entire project: What are the main findings, what are the limitations and how can the project be extended.

2

Background and Related Work

In this chapter I briefly introduce the terminology used in this work, I describe the different models of parallelism used during DNN training, and highlight relevant related work.

A DNN typically consists of one input layer, one output layer and one or more hidden layers in between (21). These layers come in different sizes, shapes and types such as linear, convolution and recurrent. As I focus on use cases related to image classification as a proof of concept, the layers of interest are linear and convolution layers. In this context, the input layer of a neural network consists of pixels from an image, each pixel being a node.

In Figure 2.1, the input layer is a linear layer type: Each node from the input layer is directly connected to all nodes in the first hidden layer, with each connection (edge) having a weight. The value of each node in the first hidden layer is then calculated by taking a dot-product over the weights of the incoming edges with the nodes in the previous layer (22). This process is repeated for each layer until the output layer is reached, resulting in a classification. This entire process is called the forward pass as activations, the values of the nodes, are propagated from the input layer to the output layer.

A neural network can 'learn' by changing the values of the weights based on the difference between the predicted and expected classification (using labeled data). The weights determine how pixel values from an image in the input layer are converted to activations in the next layer and eventually to a classification in the output layer. Gradient descent is one of the algorithms used to implement this learning process and is explained in Equation 2.1. Starting in the output layer, the difference between the predicted and expected value of each node is calculated (delta). Then for each edge connecting the output layer with the previous layer the gradient is computed, indicating how each weight should change to minimize delta, as that should result in better classifications. Finally, the gradients are

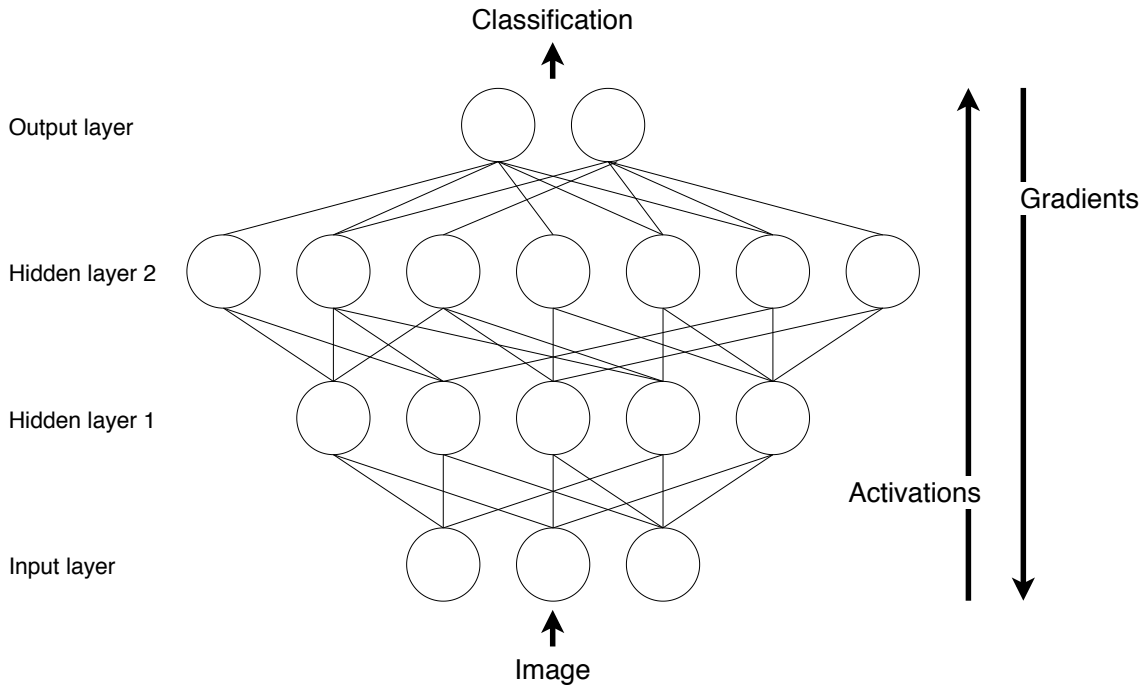


Figure 2.1: Neural network for image classification with 4 linear layers.

multiplied with a small learning rate (typically in the order of 0.01) to prevent overshooting. This is repeated for all layers, starting in the output layer and ending in the input layer, therefore called the backward pass.

$$\begin{aligned} delta &= predicted\_classification \quad expected\_classification \\ gradient &= activation_{Layer\ \ 1,i} \quad delta \\ weight &= weight \quad gradient \quad learning\_rate \end{aligned} \tag{2.1}$$

Most deep learning applications for image classification use a variant of gradient descent called stochastic gradient descent (SGD). SGD performs gradient descent on a subset of the training data at a time (called a batch), resulting in faster training times, but lower convergence rates. I use this for all applications in this thesis. As gradient descent is an iterative process, the neural network is trained multiple times on the same dataset. Here one iteration, called an epoch, consists of training on all training data once.

2.1 Terminology

Standardization is a very important process to make a meaningful comparison between scientific experiments in the same field possible. Without it, it would not only be difficult to understand how a new technique works because of the introduction of new terminology, a comparison between related papers would be unclear because of the use of custom measurements or experimentation setups. Although many papers in the field of deep neural networks and parallel programming attempt to use a uniform terminology, there is no official overarching standardization of terminology so I define the following terms in the context of this thesis for more transparency:

Distributed algorithm: Process describing how a piece of logic can be divided over multiple workers with the goal of improving performance by making use of the additional resources that the increased number of workers give. Examples of this are data, model and pipeline parallelism.

Distribution model: An instantiation of a distributed algorithm. Examples of this are Horovod, GPipe and PipeDream.

Architecture: The structure of a deep neural network, consisting of nodes and edges. Examples of this are ResNet, VGG and EfficientNet.

Model: An instantiation of an architecture with weights and an activation and loss function amongst other things.

Framework: A software implementation which enables users to easily use multiple deep learning algorithms, networks and datasets out of the box. Examples of this are PyTorch, Keras and TensorFlow.

In the next part I introduce three state-of-the-art distributed algorithms for deep learning: data-, model-, and pipeline parallelism. These methods are based on SGD, and divide the computational workload of training a model over multiple workers, while usually adding communication between those workers to enhance the learning process.

2.2 Data Parallelism

With data parallelism, each worker has a local copy of the full model and trains on a part of the data, either by partitioning or random sampling, thereby reducing the total

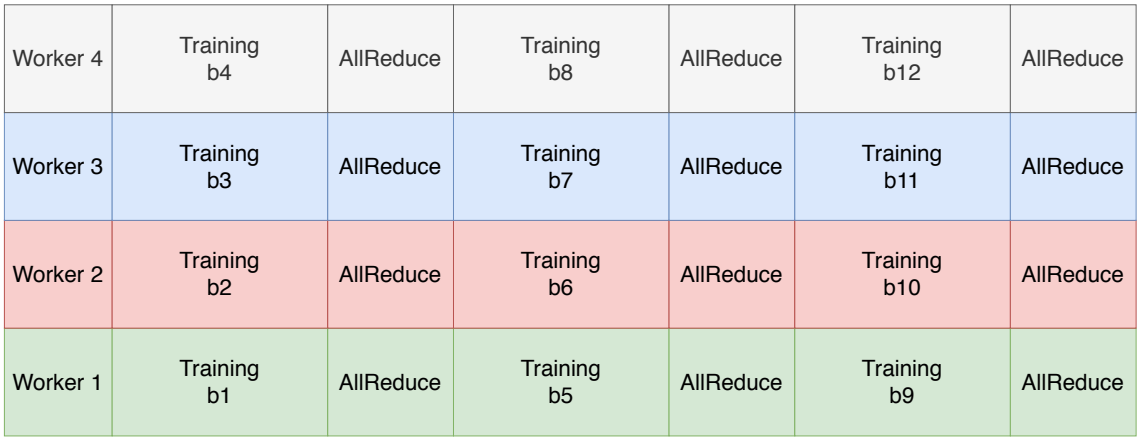


Figure 2.2: Execution pipeline of data parallelism, processing 4 batches at once (and 12 batches, b1-b12, in total).

training time. After a period of local training, the workers exchange their local gradients to calculate the global averaged gradients (using an allreduce operation), which approximates the gradients gained when using one worker to train on all data. The performance of the communication phase is critical in achieving good training time scalability with multiple workers, as the compute workload is always divided equally over all workers.

The communication phase can be implemented in a couple of ways: synchronous or asynchronous, and centralized or decentralized. With asynchronous communication, workers do not wait for each worker to finish their local training, resulting in faster communication but introducing staleness of weights as older gradients from slower nodes may be used to calculate the global average (23). Staleness introduces a loss of accuracy (24)(25), but the time-to-accuracy may not necessarily suffer as asynchronous communication is faster, resulting in lower total training times. With centralized communication, nodes are either compute nodes, which do training, or parameter servers, which gather all local gradients from the compute nodes, calculate the global average gradients and send them back to the compute nodes. With a decentralized architecture, each worker performs both roles, making better use of the available bandwidth between nodes (26), resulting in faster communication. Synchronous, decentralized data parallelism has been visualised in Figure 2.2, as all workers do local training and wait on each other to exchange gradients.

2.3 Model Parallelism

If a deep learning model becomes too large - that is, the model's parameters do not fit into the memory of a machine - data parallelism stops working because each worker requires

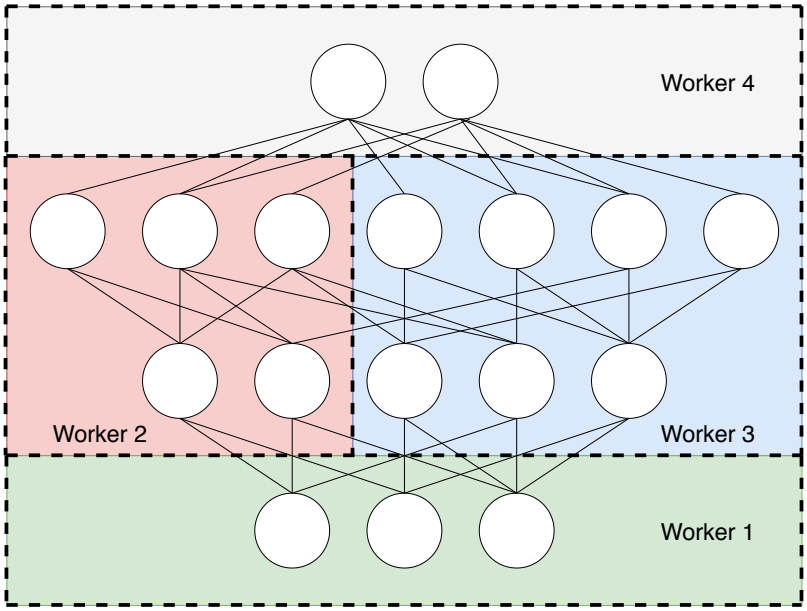


Figure 2.3: Model parallelism.

a copy of the full model. This can quickly happen on GPUs, as their memory capacity is limited compared to that of CPUs. Model parallelism solves this problem by partitioning the model over multiple workers (i.e., partitioning the model instead of partitioning the training data), thus lowering the memory usage per worker. The worker processing the first model partition uses the input images to do a forward pass on its model partition, and passes the activations of its final layer to the next worker, which repeats this process until the forward pass finishes on the worker with the final model partition. This process is then executed in reverse for the backward pass, sending gradients instead of activations (Figure 2.3).

This method creates a data dependency between workers for both the forward and backward pass, as each worker has to wait on one of its neighbours to send it activations or gradients, resulting in sequential training. Furthermore, as model parallelism introduces a communication overhead, a slowdown is accumulated with an increasing number of workers. However, model parallelism enables the use of more expressive models and larger batch sizes, compared to data parallelism due to the reduced memory usage per worker (27).

2.4 Pipeline Parallelism

The problem with model parallelism is that, on average, less than on worker is training on a part of the model because of data dependencies between workers and communication

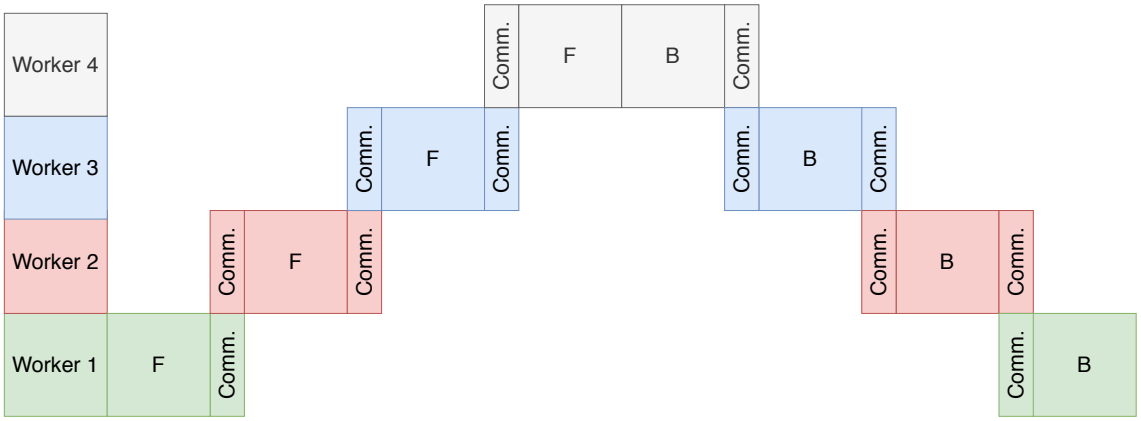


Figure 2.4: Execution pipeline of model parallelism.

overhead (Figure 2.4). Pipeline parallelism is a direct improvement on model parallelism by introducing more inter- and intra-batch concurrency via data parallelism. The most prominent pipeline distribution models for deep learning models are GPipe (1) and PipeDream (20), both using input pipelining.

2.4.1 GPipe

I focus on torchgpipe (19), a GPipe implementation in PyTorch (Chapter 3). The input pipelining that GPipe deploys is visualised in Figure 2.5. Specifically, GPipe splits each batch into micro-batches that can not only be processed in parallel, but can also be used to overlap communication with computation. Although the effectiveness of the overlapping depends on how long the communication takes compared to the compute tasks, it still results in much better scaling of the training time with the number of workers compared to model parallelism. This method does not decrease the model's accuracy, because the batch size does not change, and the weights are updated after the backward pass in a synchronous phase using the gradients of each micro-batch. To reduce the memory usage even further compared to model parallelism, GPipe does not save the activations for the backward pass but recomputes them. This is done in the 'gap' between the forward and backward passes (Figure 2.5), to reduce impact on the training time.

2.4.2 PipeDream

Where GPipe only uses intra-batch parallelism, PipeDream adds inter-batch parallelism for even faster training. This is achieved by removing the synchronization phase of GPipe after the processing of a set of micro-batches, thereby filling up the execution pipeline of each

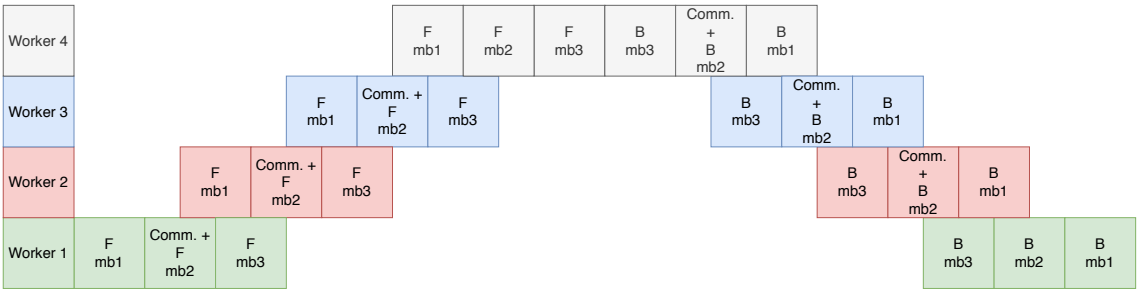


Figure 2.5: Execution pipeline of GPipe with 3 micro-batches per batch.

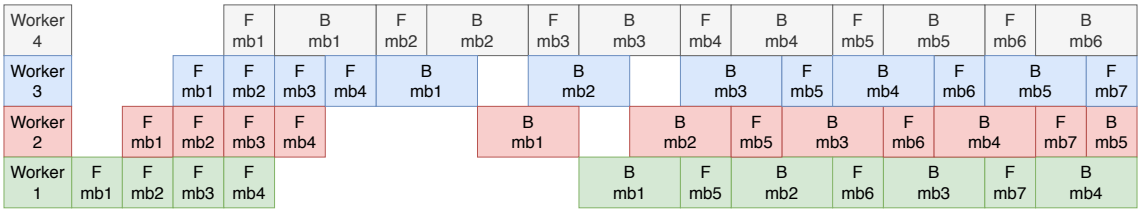


Figure 2.6: Execution pipeline of PipeDream with 4 micro-batches per batch.

GPU completely after a startup phase (Figure 2.6). The removal of the synchronization phase does introduce staleness, as micro-batches are trained on old weights. However, the overall time-to-accuracy does not suffer compared to GPipe, because the inter-batch parallelism makes training significantly faster. PipeDream saves all activations for the backward pass, because there is no time to recompute them once the execution pipeline of each GPU is completely filled.

PipeDream includes a profiler which automatically analyses a neural network to create model partitions of equal workload. However, with pipeline parallelism, this can be an impossible task because neural networks may contain a small number of layers which comprise a significant part of the workload, in which case workload imbalance ruins the performance of pipeline parallelism. PipeDream solves this by allowing model partitions to be processed by multiple workers with data parallelism, while the partitions themselves are still part of pipeline parallelism.

2.5 Related Work

Deep learning frameworks are available in many programming languages: Python, Java, C++, JavaScript and Go to name a few. Some of the most popular frameworks are TensorFlow (6), PyTorch (8), Keras (7), MXNet (28), Caffe (29) and CNTK (30), which all have support for distributed deep learning. This flexibility allows integration into

existing frameworks and lowers the entry threshold of deep learning for new scientists. The downside of this diversity is that working with deep learning may require the use of many different frameworks, each having its own syntax. In response to this, cross-framework tools such as ONNX(31) and MMdnn(32) have been created which can convert applications from one framework to another, although most conversions are experimental and need manual tuning.

Benchmark suites and infrastructures such as DAWNBench (33), MLPerf (34), and Deep500 (35) aim for standardization and generalizability, with the goal of providing a fair comparison between deep learning algorithms, frameworks, and applications. They include a variety of benchmarks, focusing not only on GPU performance, but also on CPU, interconnects, and more. However, none of these benchmarks include or compare data parallelism, model parallelism, and state-of-the-art pipeline parallelism. As these pipeline parallelism distribution models have proven to be an improvement on traditional model parallelism (1)(20), their benchmarking is vital for any generalizable distributed deep learning benchmark suite.

Analysis of the complexity, training time and memory usage of SGD and distributed deep learning algorithms have been popular research topics, although most often solely analytical (25)(36)(23)(37). Of the performance predictions tools available (27)(38), most focus on one particular use case, not rivaling the scope in terms of datasets, neural networks and distribution models of this research. Furthermore, as pipeline parallelism is a relatively new form of distributed deep learning, not much research on it has been done (39), and none focus on pipeline parallelism as implemented in GPipe and PipeDream.

3

Benchmark Suite

The recommender system [1] takes a deep neural network, a dataset and hardware characteristics and recommends the fastest distribution model to the user. For the recommender system to be generalizable, it should be able to predict training times with high accuracy for compute and communication-intensive deep neural networks, datasets of different sizes and resolutions and data, model and pipeline parallel distribution models on hardware systems with varying characteristics. For this reason I create a generalizable distributed deep learning benchmark suite which can not only be used to calibrate the recommender system with, but also to gain insight into the performance of deep learning applications.

Distribution models come in different types and programming languages. As the field of deep learning currently revolves heavily around frameworks, this is the starting point for finding distribution model candidates. The advantage of a deep learning framework is that with a few lines of code an efficient deep learning application can be created, which reduces the amount of time spent on developing non-essential functions in favor of doing meaningful research. I take a look at two popular frameworks: TensorFlow (6) and PyTorch (8) for the Python programming language.

3.1 Frameworks

Because of their popularity, many research projects have made implementations of networks or distribution models using one of these two frameworks (40). In Table 3.1 the most important distribution models for these frameworks are shown with their support of CPUs and GPUs and the distributed algorithms they implement. Only distribution models with GPU implementations have been taken into account as this is the platform I focus on, but

[1] Submitted to NeurIPS

Table 3.1: A comparison of different distribution models for machine learning.

Distribution model	*TF*	*PyTorch*	*CPU*	*GPU*	*Data*	*Model*	*Pipeline*
tf.distribute	X		X	X	X		
Mesh	X		X	X		X	
PipeDream		X		X		X	X
GPipe	X	X		X		X	X
Horovod	X	X	X	X	X	X	
torch.distributed		X	X	X	X	X	

the benchmark suite can be extended to support CPU implementations as well. CPUs may not have the computing power of GPUs, but do have the advantage of more memory capacity.

While Tensorflow and PyTorch both support the same programming language and syntactically may look similar, supporting both frameworks is out of the scope of this project. The most suitable framework for this project is the one that supports the most diverse set of distribution models, that is at least support for data and pipeline parallelism. Supporting diverse distribution models will maximize the chance that a truly performance optimal distribution model is chosen by the recommender system instead of a suboptimal one because of a scarcity of choice.

For data parallelism one can choose between tf.distribute and Horovod for Tensorflow and between torch.distributed and Horovod for PyTorch. Horovod has been the better performing version for both platforms as it uses decentralized data parallelism compared to tf.distribute and torch.distributed which mainly use centralized data parallelism, which performs worse in most situations (36). However Tensorflow has updated its distribute package to support decentralized data parallelism as well, potentially rivaling Horovod's performance. In PyTorch this is not supported out-of-the-box although it is possible to implement ring allreduce using point-to-point communication.

Tensorflow Mesh is the only distribution model which focuses on model parallelism, while PipeDream and GPipe can disable the pipeline parallelism optimizations to get model parallelism and Horovod and torch.distributed use data parallelism features to simulate model parallelism. As both pipeline parallelism distribution models support model parallelism, I do not select a separate distribution model for model parallelism.

For pipeline parallelism the prime candidates are PipeDream, which has PyTorch support, and GPipe, which has Tensorflow support but was ported to PyTorch in (19). Although both implement pipeline parallelism, how they implement it is very different so supporting both is highly preferred. As Horovod covers data parallelism for Tensorflow and PyTorch and both PipeDream and GPipe are available in PyTorch, I choose for PyTorch as it supports more diverse distribution models than Tensorflow with Horovod, PipeDream and GPipe.

3.2 Datasets

I start with selecting datasets as neural networks are often specifically made for a certain dataset and so depend on the choice of datasets. I use (41) to find the most used datasets and networks per use case for published research papers. In the case of image classification the most popular datasets are ImageNet, CIFAR-10, CIFAR-100 and MNIST. The CIFAR datasets contain 60000 images divided over 10 and 100 classes respectively. I do not use the CIFAR-100 dataset for the benchmark suite as it is very similar to the more popular CIFAR-10. MNIST (12) contains images of handwritten numbers between 0 and 9. The images in CIFAR-10 (13) are divided over 10 classes such as airplane, bird, cat and ship. ImageNet-1000 (14) is a subset of ImageNet and contains 1000 classes such as animal species, food and furniture. Although the images in this dataset have varying resolutions, they are all resized to 224 by 224 pixels for training as this gives uniformity which accelerates the learning process. These three datasets are included in PyTorch. I add a synthetic dataset with high resolution images, called Highres, to study the effects of models with large memory footprints on training time (Table 3.2).

Table 3.2: Selected datasets for image classification.

Name	*#classes*	*#images*	*Color profile*	*Resolution*
MNIST	10	70000	Grayscale	28 x 28
CIFAR-10	10	60000	RGB	32 x 32
ImageNet	1000	1280000	RGB	224 x 224
Highres	1000	60000	RGB	512 x 512

3.3 Networks

The final selection process is about choosing the right networks for each dataset. Each network has its own characteristics which fit some datasets better than others, resulting in a higher accuracy or lower training time. However, as the goal of the benchmark suite is to calibrate the recommender system, I use the same set of networks for each dataset as this enables a meaningful performance comparison between datasets and models. I select neural networks based on the amount of compute and communication workload they create, as a change in these workloads can uncover how the training time scales on different distribution models with the number of workers.

Generalizability is achieved in the selection of distribution models by selecting models which cover all relevant distributed algorithms for deep learning. For datasets this is achieved by selecting datasets based on the number of images they contain and the dimensions of the images as this influences both the communication and computation patterns of a DL application. For networks this is very similar: Some are communication heavy while others are compute heavy in a distributed setting based (42). For compute heavy networks I choose ResNet (16) and for communication heavy networks VGG (17) as those are one of the most popular family of networks. Furthermore, they come in different sizes: ResNet-18 to ResNet-152 and VGG-11 to VGG-19 are included in PyTorch. With this it is possible to simulate different intensities of computation, communication and memory usage as the networks grow in size.

To these two network families of networks I add MobileNet v2 (15) as it is a neural network targeting the mobile market. As mobile phones have an energy budget and limited resources, these networks have different characteristics compared to regular networks like ResNet and VGG such as different types of layers.

For ResNet I use the 18, 50 and 152 layers variants as the smallest, most popular and largest ResNet versions respectively. For VGG I select the 11 and 16 layers variants as the smallest and most popular versions respectively. I do not select the largest network there, VGG-19, as there is too little difference from VGG-16. More information on the networks can be found in Appendix A.

For GPipe and PipeDream the models have been slightly altered as GPipe demands a list of neural network layers instead of the usual class-based representation of PyTorch and PipeDream does not support all versions of the neural network layers in PyTorch. However this does not affect performance in any way as the contents of the models are still the same.

The validation of the ImageNet-1000 models can be found in the PyTorch documentation. For CIFAR-10 this can be found in (43). The MNIST models have been verified using the benchmark suite itself, resulting in accuracies as high as 99 percent. This is not surprising as the MNIST dataset is small and easy to learn while the selected models are large and expressive. All ImageNet and VGG models have been verified using the benchmark suite as well, confirming the accuracies of previously mentioned sources.

3.4 Experimental Setup

As I focus on training speed and not on accuracy, I use seconds per epoch as the main performance metric. I also report speedup, for which I use PyTorch with 1 GPU as baseline. The storage location of the training data is critical for training speed as storage in a central location could lead to network contention, resulting in fluctuating training times (44). Although it would be interesting to analyse how the different distribution models perform when directly reading the data from a central storage, it is not the focus of this research and so I write synthetic versions of the datasets (Table 3.2) to the local storage of each node. As there are no other factors which can cause major fluctuations in the training time per epoch, I limit the number of epochs to 3 (this is configurable), and report the average training time per epoch.

3.4.1 Hardware

For all experiments I use a cluster of NVIDIA Titan RTX GPUs with 24 GB of device memory. Each node has 4 GPUs, two pairs connected with NVLink, and dual-socket Intel Xeon Gold 5118 CPUs with 192 GB of RAM. The nodes are connected via 40 Gbps Ethernet. All nodes use GCC 8.3.0, CUDA 10.1.243, cuDNN 7.6.5.32, OpenMPI 3.1.4 and NCCL 2.5.6.

3.4.2 Batch Size

An increase in batch size affects memory usage as it leads to larger activation and gradient memory sizes as more images are being trained on at once. This makes choosing a batch size an application-specific optimization rather than a generalizable one, which results in hyperparameter tuning for optimal performance. However, this is out of the scope of this project so I choose one batch size per dataset per distribution model (Table 3.3).

As Horovod does not split the model over multiple workers, the memory usage per worker does not depend on the number of workers used. The batch sizes for the Horovod

Table 3.3: Batch size configurations.

Dataset	*PyTorch*	*Horovod*	*GPipe (#micro-batches)*	*PipeDream*
MNIST	128	128	3072 (24)	128
CIFAR-10	64	64	2048 (32)	64
ImageNet-1000	32	32	384 (12)	32
Highres	32	32	48 (12)	32

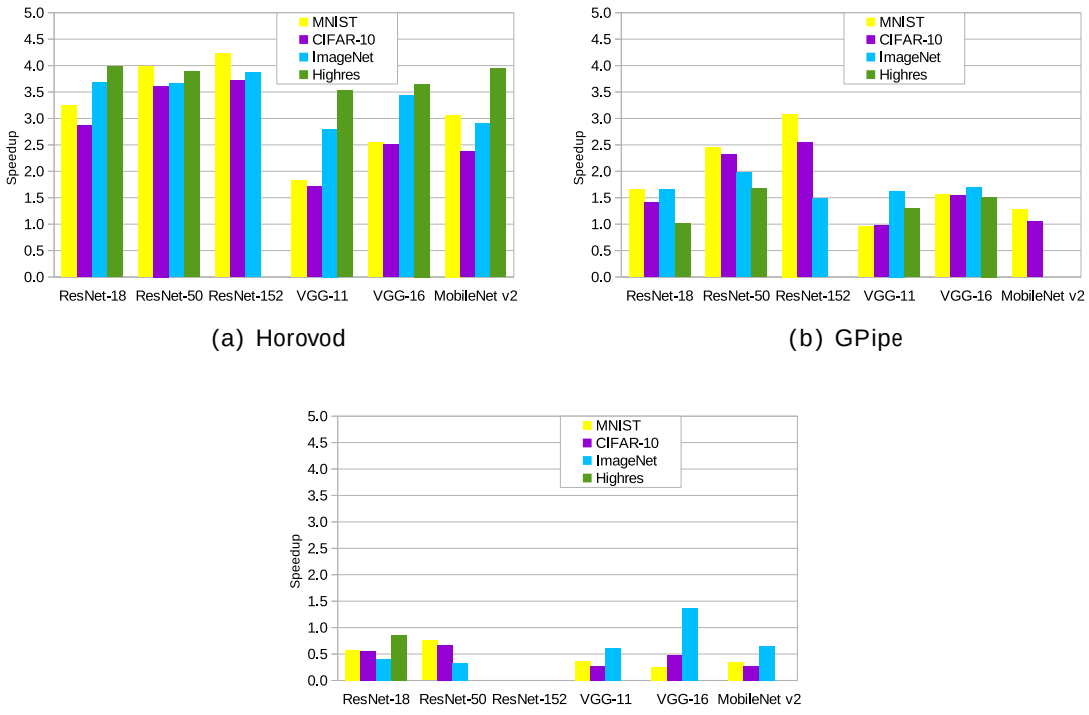
benchmarks are reported per worker, resulting in a total of $batch\_size \quad workers$ images being trained on across all workers.

GPipe and PipeDream do split the model over multiple workers however, resulting in less memory usage per worker, which can then be used to support higher batch sizes. As torchgpipe (19) does not support multi-node training, I report the batch sizes used in a 4 GPU setup. PipeDream does support multi-node training, however because of issues with its automatic profiler the same batch sizes as for PyTorch are used. The profiler performs analysis on a neural network to translate it to a PipeDream-specific intermediate representation. This analysis is performed on a single GPU, so if either the neural network is too large or the batch size to high, the analysis can not be performed due to the limited memory capacity of a single GPU. GPipe also includes a profiler for automatic model partitioning, however as it performs profiling on one layer at a time (in contrast to the whole model at once for PipeDream), GPipe is able to perform profiling on models which otherwise would not fit into the memory of a single GPU.

While a higher batch size generally results in lower training times, it also affects the convergence rate as larger batch sizes tend to result in lower convergence rates, although this can be compensated by increasing the learning rate (45).

3.5 Results

First, I perform all benchmark on a single node (4 GPUs). All selected datasets, neural networks and distribution models are being used (Figure 3.1). Some experiments did not succeed due to being out of memory (ResNet-152 with PyTorch and Horovod), and problems with GPipe (MobileNet v2 for ImageNet and Highres) and PipeDream (ResNet-152, most models for Highres). This behavior suggests a lack of maturity for pipeline parallelism frameworks, when compared to established, data-parallel ones, such as Horovod.



(a) Horovod

(b) GPipe

(c) PipeDream

Figure 3.1: Training time speedup compared to PyTorch with 1 GPU using 4 GPUs.

Horovod (Figure 3.1a) performs very well, even achieving a super-linear speedup with MNIST and ResNet-152 as each GPU processes only a quarter of the data, and therefore can cache more data. Furthermore, ResNet performs better than VGG networks on average as there is relatively more compute work for the former, resulting in better scalability. The smaller MNIST and CIFAR-10 datasets perform worse for VGG than the larger ImageNet and Highres datasets as there is less compute work to overlap with the communication (Chapter 5).

GPipe (Figure 3.1b) follows the same trends as Horovod: Good scalability for ResNet (except for the smaller ResNet-18) and much less performance increase for VGG. The performance for ResNet is similar to that of (19) and (1). GPipe should achieve better scalability for VGG compared to ResNet than Horovod as only a small part of the activations and gradients are communicated with GPipe instead of all gradients with Horovod, however VGG performs as bad for GPipe as for Horovod, indicating that the implementation of the communication may not be as optimal as predicted (Chapter 6).

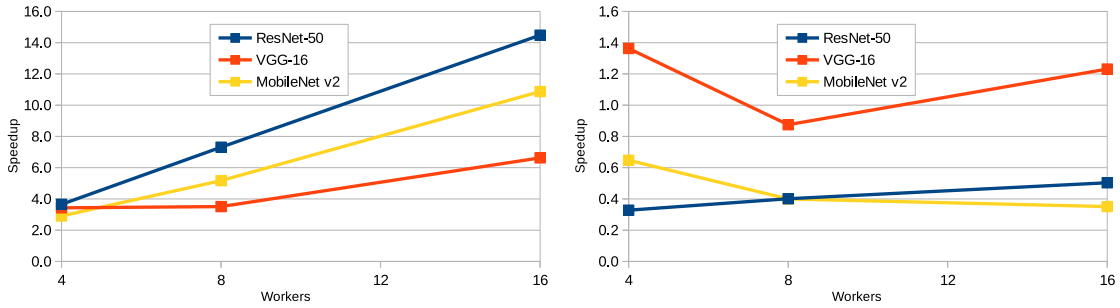
PipeDream (Figure 3.1c) performs much worse than GPipe for all benchmarks, most likely due the usage of much smaller batch sizes. However only the difference in batch

sizes can not explain this discrepancy: I take an in-depth look in Chapter 7. Only the benchmark with ImageNet and VGG-16 achieves a speedup compared to the baseline, most likely due to using full data parallelism, although in that case the speedup still is nowhere close to that of Horovod.

3.5.1 Multi-Node Scalability

Only for Horovod and PipeDream multi-node experiments with 1, 2 and 4 node have been conducted as torchgpipe does not support multi-node execution (Figure 3.2). For Horovod there are no surprises: The compute intensive ResNet-50 performs the best, achieving a 14.5 speedup with 16 workers, while VGG-16 performs the worst as communication quickly dominates execution time.

PipeDream on the other hand shows a slowdown with an increasing number of workers, with only the VGG-16 benchmark achieving a speedup over the sequential baseline. This is due to a combination of compute and communication problems as both ResNet-50 and VGG-16 do not show any good scaling (Chapter 7).



(a) Horovod (b) PipeDream

Figure 3.2: Training time speedup for ImageNet compared to PyTorch with 1 GPU using 4, 8 and 16 GPUs.

4

Performance Prediction

A distribution model divides the compute work of training a model over multiple workers and adds communication between workers to exchange data, be it activations, weights or gradients. As data, model and pipeline parallelism do the same amount of training per epoch as a sequential deep learning application, I start by predicting the execution time of such a sequential application. This will later on be used as a basis for the performance models of Horovod, GPipe and PipeDream.

For all experiments I use the benchmark suite described in Chapter 3 as a baseline to calibrate the performance predictor, All datasets, neural networks, distribution models, and hardware configurations from the benchmark suite are used for the experiments.

4.1 Statistical Model

Analytically modeling the performance of (distributed) deep learning applications has been a very popular topic in the last two-three years (46)(27)(39). However, it remains difficult to accurately predict the performance of DL applications with such models due to the complexity of both DL workloads *and* hardware systems. Even if one could build an analytical model which accurately predicts the number of operations needed to train a model, converting this to execution time would be difficult, as the number of FLOPS a CPU or GPU can theoretically output is vary rarely achieved by an application.

To avoid these pitfalls, in this work, I use a statistical modeling approach: the DNN model is trained on a small number of batches from the dataset, whereafter this training time is extrapolated to the training time for all batches of the dataset. The training takes place on the same hardware the application that is predicted for will make use of to guarantee as accurate predictions as possible. However only a single GPU is needed

as the statistical modeling approach is used to predict the performance of a sequential DL application. This method is expected to be more accurate than an analytical model, as the DL application is executed instead of modeled. Nevertheless, this method faces three obstacles. First, it takes more time to train the model on any amount of data than solving an analytical formula. However, as the number of data samples trained on per second is relatively stable throughout an epoch as the same actions are performed for each data sample on the same hardware, the training overhead is only in the order of seconds to guarantee accurate extrapolation results (Figure 4.1). Second, as the predictor models the behaviour of a DL application on certain hardware, the same piece of hardware should be used to perform to performance prediction on. In principle, this should not be a showstopper as the predictor only requires a single GPU and a predictor's user most likely has access to the infrastructure targeted for the actual training. Third, and final, the statistical model cannot predict the performance of models which are too large to fit into the memory of a single GPU. One could train the model on a CPU instead of a GPU for the statistical model, as those tend to have higher memory capacity, but the processing speed of a CPU differs from that of a GPU, which makes getting an accurate prediction complicated. Making this conversion is an interesting direction for future work, thus enabling this method to support models with a large memory footprint, but it is outside the scope of this work.

Table 4.1: Information passed from the user to the recommender system

Name	*Description.*
Model	The deep neural network.
Training data	All data needed to train the model.
#Nodes	Number of servers on which the application is executed.
#GPUs per node	The number of GPUs available per server.
Intraconnect speed	The throughput in Megabytes per second between GPUs on the same node (optional).
Interconnect speed	The throughput in Megabytes per second between nodes (optional).
Gpipe model	The DNN used for GPipe, as it may differ from the regular DNN.
#GPipe micro-batches	The number of micro-batches a batch is split into for GPipe.

4.2 Configuration

The information that is passed to the recommender system by the user is displayed in Table 4.1. Note that the recommender system is always executed on one node, the number of nodes in Table 4.1 relates to the number of nodes the DL application that is predicted for will run on. Only the model and training data are needed for the prediction of the sequential execution time, all other variables are explained in later chapters.

4.3 Results

Figure 4.1 displays the relative difference between the predicted execution time in seconds per epoch and the actual execution time from the benchmark suite. Here a positive difference signifies an overestimation of the predictor and a negative difference an underestimation. The predictor is at most 15 percent off, primarily due to the training time per batch not being completely constant over an entire epoch. This deviation is small enough to give users accurate estimations of the execution time of a full DL application, no matter the dataset and neural network used. The prediction for Highres with ResNet-152 is missing due to memory limitations (Chapter 3).

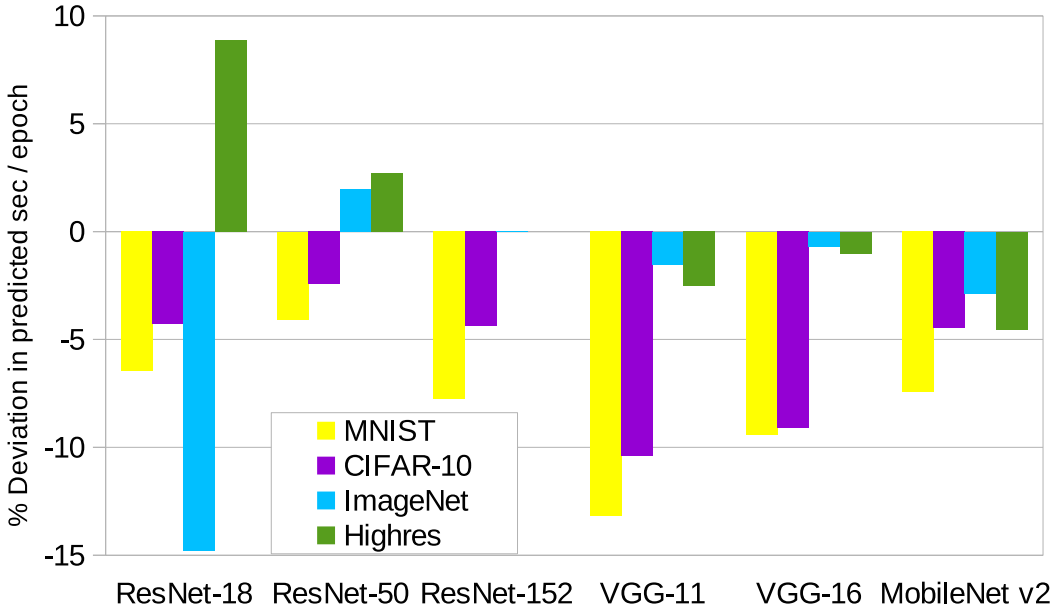


Figure 4.1: Accuracy of the performance predictor for sequential DL applications.

5

Horovod

To speedup the training process, Horovod uses multiple workers and data parallelism with ring-allreduce synchronization(18). The compute workload is composed of a forward and backward pass for each batch in a dataset, while the communication workload is composed of exchanging gradients with one or multiple ring-allreduce operations (Chapter 2).

5.1 Performance Model

The performance model for Horovod can be found in Equation 5.1; the corresponding list of symbols is included in Table 5.1. This performance model predicts the training time per batch, which is supposed to be constant for all batches in one epoch. The training time per epoch is then computed by multiplying the training time per batch with the number of batches in a dataset. With data parallelism, the compute workload of a sequential DL application is equally divided over multiple workers, resulting in a performance model for the compute workload of $\frac{T_{seq}}{W}$.

$$T_{horovod} = \frac{T_{seq}}{W} + 2(W \quad 1) \quad \max_{i=1}^{W}(L_{i,i+1} + \frac{\min(G, th)}{W \quad BW_{i,i+1}}) \tag{5.1}$$

5.1.1 Ring-AllReduce

Horovod uses the AllReduce operation to exchange gradients between workers. Although the total number of gradients exchanged between workers per batch is constant ($W \quad G$), multiple communication schemes are possible in Horovod, each with its own number of AllReduce operations and number of gradients exchanged per AllReduce. A first scheme does one AllReduce operation per batch, after each worker has completed the entire backward pass, with $W \quad G$ gradients being exchanged at once. A second scheme performs many

Table 5.1: List of symbols

Name	*Unit*	*Explanation*
T_{seq}	sec	Execution time of a DL application using 1 GPU
W		Number of workers
$L_{i,j}$	sec	Latency from worker i to worker j
G	MB	Size of all gradients in a model
th	MB	Maximum amount of data to be processed per AllReduce
$BW_{i,j}$	MB/sec	Bandwidth from worker i to worker j
mb		Microbatch size
b		Batch size
A_i	MB	Size of all activations in model-partition i
W_i	MB	Size of all weights in model-partition i

AllReduce operations per batch, namely one per layer processed in the backpropagation. It requires all workers to synchronize after each worker has processed a single layer locally. This results in $\#layers$ AllReduce operations of $W \quad G_i$ gradients (i being the index of the layer). The advantage of using this scheme over the first one is that it is possible to overlap all AllReduce operations (except for the final one) with computation, resulting in less communication overhead. However, as this scheme results in many small messages to be sent between workers, it becomes easily bottlenecked by latency.

Horovod adds a third, hybrid communication scheme, which combines the best of both worlds: it does an AllReduce operation each X milliseconds, with up to Y MB of gradients. The maximum amount of gradients in MB allowed per AllReduce operation is called the threshold (th). These two parameters (X and Y) are configurable, and Horovod provides a script to automatically tune them for optimal performance. This method is called *Tensor Fusion*, as it can fuse the gradients of multiple layers together into one AllReduce operation. It eliminates the latency bottleneck while still overlapping computation with communication. Note that AllReduce operations with more than Y MB of gradients at a time can still occur, if a single layer has more gradients than the threshold value.

The resulting communication overhead of tensor fusing depends only on the final AllRe-

duce operation, assuming all other AllReduce operations can be overlapped with computation. However, predicting how many gradients are included in this final AllReduce operation is extremely difficult, because the tensor fusing algorithm is very dynamic: each X milliseconds, an AllReduce operation takes place, so any small deviation in the execution time of the backpropagation of one layer can change the scheduling of all following AllReduce operations completely. Because of this behaviour, I chose to use a conservative assumption in the predictor: the amount of gradients participating in the final AllReduce operation is equal to the threshold, if the total amount of gradients in the neural network is at least the size of this threshold. In other words, I assume $\min(G, th)$ MB of gradients participating in the final AllReduce operation. This is equivalent to the worst-case scenario.

The AllReduce operation can be implemented in different ways, depending on which backend is used (MPI, GLOO, or NCCL for PyTorch). These are most often based on broadcasts, rings, or trees. Although in (18), the Ring-AllReduce operation is explained in detail for Horovod, the actual AllReduce algorithm being used at runtime may differ. As the predictor cannot find out which AllReduce algorithm is used by the backend at runtime, I assume Ring-AllReduce is used. Each Ring-AllReduce operation consists of $2(W \quad 1)$ iterations, with each worker sending $\frac{min(G,th)}{W}$ MB of gradients per iteration. With this, the total amount of gradients sent per Ring-AllReduce operation per worker in MB is $2(W \quad 1) \ \frac{min(G,th)}{W}$. This estimate will eventually lead to the second, communication-related term of the performance model (see Equation 5.1).

5.1.2 Latency and Throughput

To convert the amount of data sent in MB to time in seconds, I assume a simple communication model, where the time taken to transmit B bytes from worker i to worked j, is composed of the latency of starting the communication, L_{ij}, and the ratio between the data size and the link bandwidth, i.e., $\frac{B}{BW_{ij}}$.

However, to fully estimate this communication time, knowledge of the latency and bandwidth between each pair of GPUs in the ring is required. I use p2pBandwidthLatencyTest, a tool created by NVIDIA (47), to obtain this values. The tool performs microbenchmarks to get the latency and bandwidth between all pairs of GPUs on a single node, and supports PCIe, NVLink, and NVSwitch connections.

Note that the user must inform the recommender system of the interconnect speed (Ethernet, InfiniBand) if predicting for more than one node, because the recommender system uses only one node for prediction, and therefore it does not perform a microbenchmark

to get the inter-node interconnect speed (Table 4.1). It is possible for the user to inform the recommender system of the "intraconnect" speed (i.e., the connection speed inside the node), which is a useful feature when using non-NVIDIA hardware.

I also note that only the slowest connection in the ring of the Ring-AllReduce operation needs to be taken into account, because this link bottlenecks the entire communication process. This results in the performance model in Equation 5.1 which predicts the training time per batch in Horovod.

5.2 Model Evaluation

I compare the execution times gained from the benchmarks in the benchmark suite to those gained from performance model (Figure 5.1). The relative difference between those two sets of training times indicates how accurate the performance model can predict the execution times of real-world distributed deep learning applications.

The results show that the predictions are at most 15% off for 4 out of 6 neural networks; the exceptions are VGG-11 and MobileNet v2 (Figure 5.1). These small deviations can be explained by the deviations in the predicted sequential execution time (Figure 4.1), and the final AllReduce operation not using the maximum amount of data possible, which I assumed in the design of the predictor.

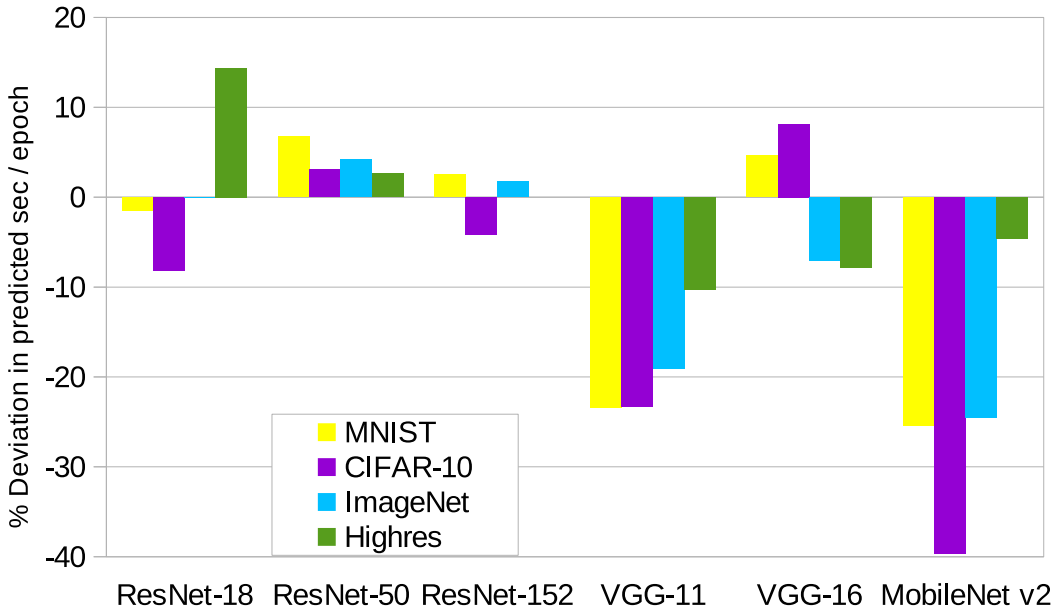


Figure 5.1: Accuracy of the performance predictor for Horovod using 4 GPUs.

For VGG-11 there is very little compute work to do with only 9 convolutional and linear layers for MNIST and CIFAR-10, and 11 for ImageNet and Highres. Meanwhile, the total number of gradients, and therefore the communication overhead is high (42), resulting in communication not being perfectly overlapped with computation for all AllReduce operations. The current performance model assumes that all AllReduce operations, except the final one, are perfectly overlapped with computation. Because this is not the case for VGG-11, *the model is optimistic*, and predicts too low training times. The performance model can be improved in the future by taking the training time for each layer during the backpropagation into account, together with the number of gradients per layer, to determine if communication can be completely overlapped. However, as the scheduling of the AllReduce operations differs greatly between batches, the success of this approach is not guaranteed.

To understand what happens in the case of MobileNet v2, I use the PyTorch Bottleneck tool to profile the training process of the MobileNet v2 benchmarks. The tool keeps track of all PyTorch and backend (GLOO, NCCL) function calls, for each thread and GPU, resulting in a time series dataset. MobileNet v2 has such a small number of gradients that the predicted communication overhead is negligible. However, the PyTorch Bottleneck tool shows that the communication overhead is in fact significant. This is because I only take the communication part of the Ring-AllReduce operation into account; however, communication also includes a negotiation phase, memory buffer allocation, and a memory copy phase. The time these three phases take up is much less dependent on the amount of data that is communicated than the communication subphase, so for a Ring-AllReduce operation where little data is sent, these three phases start to dominate the execution time. In the case where more data is sent per Ring-AllReduce, the performance model slightly overestimates the communication time to take the negotiation, memory buffer allocation and memory copy phases into account, however for MobileNet v2 there is so little data to be sent that the model can no longer compensate, resulting in a loss of accuracy. Another factor that comes into play is load imbalance. For the negotiation phase to start, all workers should finish doing the backpropagation up to the same layer. Although in theory all workers should finish their backpropagation at the exact same time as they have the same amount of training data and use the same neural network, in practice the training speed differs slightly between GPUs, resulting in a load imbalance overhead. This overhead is usually so small that for Ring-AllReduce operations with an average amount of data the overhead can be ignored, however for the small AllReduce operations such as with Mo-

bileNet v2 this is more significant. Currently, the performance model does not take either load balancing or the negotiation phase into account as it is very difficult to predict for.

Overall, all predictions for MobileNet v2 are optimistic. However, the prediction for the Highres dataset is much more accurate than the predictions for the other three datasets. One of the possible explanations is that the computational workload per batch is much higher for Highres compared to that of MNIST, CIFAR-10 and ImageNet, because the image dimensions are higher. Thus, the impact of the communication overhead on the overall execution time is significantly reduced.

5.2.1 Multi-Node

I also experiment with multi-node predictions for Horovod using the benchmarks from Chapter 3. Again, the analysis focuses on the deviation between predicted and measured execution time. The results are presented in Figure 5.2.

First, note that the results demonstrate good accuracy for ResNet-50, but worse accuracy for VGG-16 and MobileNet v2.

Moreover, for ResNet-50 and MobileNet v2, the accuracy of the predictions does not change when increasing the number of workers, while the accuracy of the VGG-16 benchmarks drops significantly from 4 to 8 workers. This drop cannot be caused by a decrease in computational efficiency, because each worker still has the same amount of training data to process. However, in the communication phase something changes: as the number of workers increases, the amount of data sent per step per worker in the Ring-AllReduce algorithm decreases. This results in the same problem as with MobileNet v2 in the single-node benchmarks: the communication part of the Ring-AllReduce operation takes up so little time, that parts like negotiation and memory allocation start to dominate the execution time of the AllReduce operations, which is not taken into account in the performance predictor.

This explanation is supported by the results in Figure 5.2: for ResNet-50 the accuracy does not change as the number of gradients to communicate is insignificant compared to the computational workload, while for both VGG from 8 to 16 GPUs and for all MobileNet v2 results the accuracy does not change as the Ring-AllReduce operations are already dominated by the negotiation and memory allocation phases.

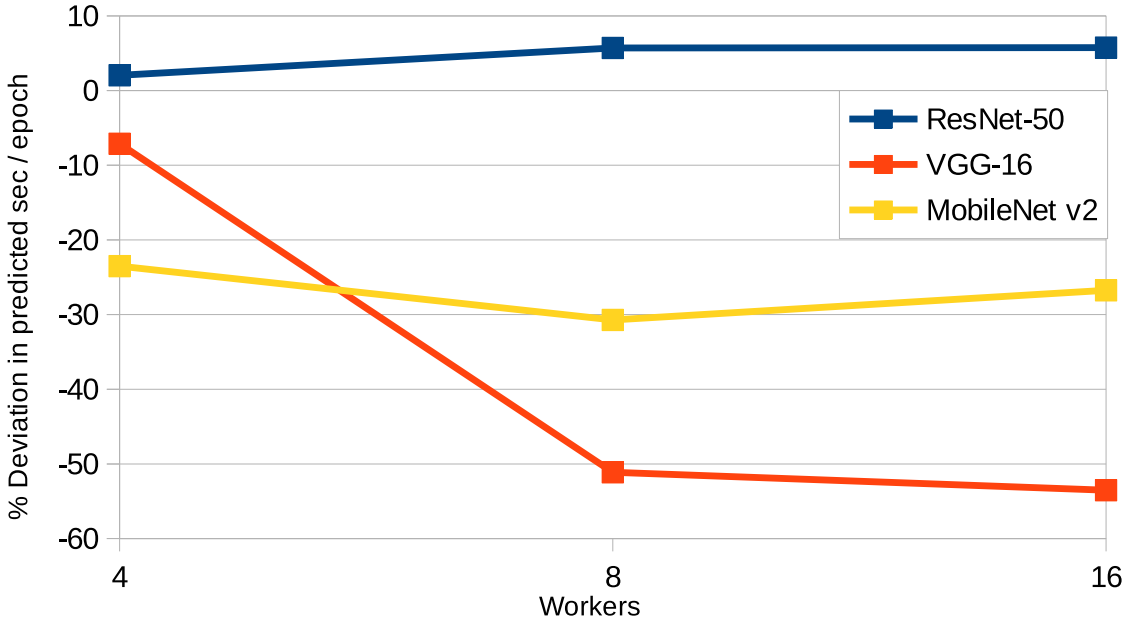


Figure 5.2: Accuracy of the performance predictor with ImageNet for Horovod using 4, 8 and 16 GPUs.

5.3 Summary

The performance predictor for Horovod can accurately predict the execution times of compute intensive benchmarks with large amounts of data to be communicated between workers. There are two limitations to the predictor: First, the prediction accuracy for deep learning applications dominated by communication is low as the predictor assumes that all communication except the final Ring-AllReduce operation can be overlapped with computation. Second, the prediction accuracy is also low if the amount of data sent per worker per Ring-AllReduce operation is too little, as the negotiation phase between workers start to dominate the Ring-AllReduce operation time, which is very difficult to predict for and so not included in the performance model.

6

GPipe

With model parallelism, the layers of a neural network are divided over multiple workers, thereby reducing the memory usage per worker. GPipe uses pipeline parallelism to speedup model parallelism by introducing input pipelining: each batch is split into multiple (smaller) micro-batches, which can be processed in parallel. As the framework of choice in this project is PyTorch, I use a variant of GPipe specifically build for PyTorch called torchgpipe (19).

6.1 The Computation Performance Model

A visualisation of the workload of model parallelism is given in Figure 2.4: the first worker executes a forward pass on the first part of the model, and passes the activations of its final layer to the "next" worker, which has the second part of the model. This is repeated until the worker with the last model partition executes a forward pass; afterwards, this whole process is executed in reverse, for the backward pass.

By splitting batches into multiple micro-batches, GPipe speeds the process up, as multiple workers can execute forward or backward passes on micro-batches in parallel. The principle is visualised in Figure 6.1, while equation 6.1 models the speedup. Note that, by using only one micro-batch, this performance model defaults to just T_{seq}, which is in line with the performance of model parallelism because there is no concurrency. Also note that the sequential computation time in Equation 6.1 is predicted using the size of one micro-batch as batch size, not the batch sizes that are reported in Table 3.3 for the PyTorch baseline.

$$T_{comp} = T_{seq} \quad \frac{W + \#mb \quad 1}{W \quad \#mb} \tag{6.1}$$

The intuition behind the model in Equation 6.1 is as follows. First, it takes W 1 steps to fill and empty the execution pipeline of GPipe, with a step being the execution of a forward or backward pass on one micro-batch (Figure 6.1). The number of steps where all workers are executing forward or backward passes in parallel is $\#mb$ $W + 1$. As the number of inefficient steps, where not all workers are executing forward or backward passes in parallel, only depends on the number of workers while the number of efficient steps also depends on the number of micro-batches, a higher micro-batch to worker ratio should result in better concurrency and so better performance (1).

6.2 The Communication Performance Model

GPipe relies on the overlapping of communication with computation for good performance, similar to Horovod. The communication of activations or weights to the next worker can be overlapped by the forward or backward pass on the next micro-batch (Figure 2.5). However, on closer inspection with the PyTorch Bottleneck profiling tool, torchgpipe (19) does not hide communication (Figure 6.1): computation and communication are implemented as separate phases, and therefore are not overlapped.

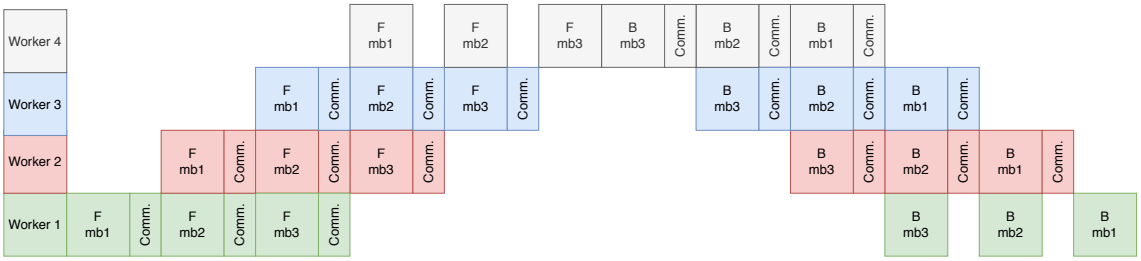


Figure 6.1: The actual execution pipeline of torchgpipe: an example with 3 micro-batches per batch. The pipeline as presented in GPipe (1) can be found in Figure 2.5.

As the workers execute in lockstep, only the slowest communication link per step needs to be taken into account. This is true for both the first and last W 2 steps, in which the execution pipeline is filled and emptied, as well as for the $\#mb$ $(W$ 2) steps in between, where the pipeline is completely filled and all workers train in parallel. As the number of activations sent in the forward pass equals the number of weights sent in the backward pass, the communication overhead of the forward pass equals that of the backward pass. The resulting model can be found in Equation 6.2, which mirrors the execution pipeline from Figure 6.1.

$$
\begin{aligned}
T_{comm} = 2 \quad & \Big(\sum_{i=1}^{W \quad 2} \big(\max_{j=1}^{i} (L_{j,j+1} + \frac{mb \quad A_j}{BW_{j,j+1}}) \big) + \\
& (\#mb \quad (W \quad 2)) \quad \max_{j=1}^{W \quad 1} (L_{j,j+1} + \frac{mb \quad A_j}{BW_{j,j+1}}) + \\
& \sum_{i=2}^{W \quad 1} \big(\max_{j=i}^{W \quad 1} (L_{j,j+1} + \frac{mb \quad A_j}{BW_{j,j+1}}) \big) \Big)
\end{aligned}
\tag{6.2}
$$

I compare the execution times gained from the benchmarks in the benchmark suite to those gained from performance model (Figure 5.1). The relative difference between those two sets of training times indicates how accurate the performance model can predict the execution times of real-world distributed deep learning applications.

I also experiment with multi-node predictions for Horovod using the benchmarks from Chapter 3. Again, the analysis focuses on the deviation between predicted and measured execution time. The results are presented in Figure 5.2.

6.3 Model Evaluation

I perform the same experiments with GPipe as performed with Horovod: I compare the execution times gained from the benchmarks in the benchmark suite to those gained from performance model (Figure 6.2). The relative difference between those two sets of training times indicates how accurate the performance model can predict the execution times of real-world distributed deep learning applications. I use the batch size and micro-batch sizes reported in Table 3.3. The results show the predictor is very inaccurate for GPipe, being off by as much as 76%.

This strange behavior can be explained by Figure 6.3, showing the output of the previously mentioned PyTorch Bottleneck profiling tool. I note that the communication phases *between* the processing of micro-batches in the forward pass take almost as long as the computation part of the forward pass; even without these communication phases, the forward pass takes up more time than the backward pass, while the backward pass contains more computational work and so should take longer (22).

On the other hand, the backward pass is executed by PyTorch autograd, PyTorch's backpropagation engine, with high efficiency compared to the forward pass, so the inaccuracy of the predictor is most likely due to an inefficient implementation of the forward pass in torchgpipe rather than an incorrect performance model. After all, GPipe should have much less communication overhead than Horovod as only activations or weights of a single

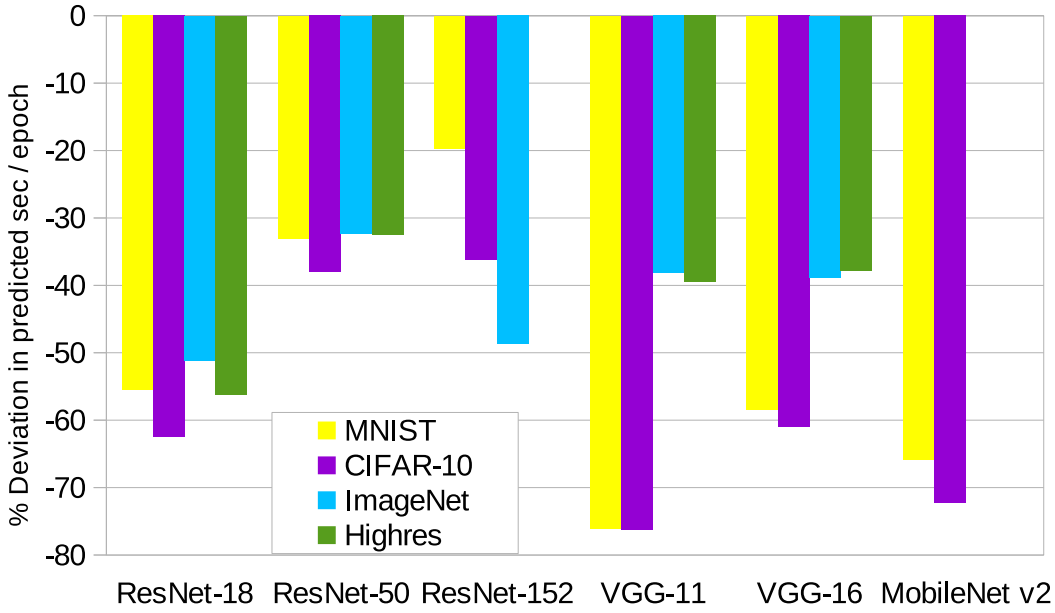


Figure 6.2: Accuracy of the performance predictor for torchgpipe using 4 GPUs.

Figure 6.3: Analysis of the PyTorch Bottleneck profiling tool on a GPipe application using ResNet-18 with CIFAR-10 on 4 GPUs with 8 micro-batches. First the forward pass is executed, then the backward pass. Only compute-related workload is shown.

layer are communicated instead of all gradients, and with the high number of micro-batches (especially for the MNIST and CIFAR-10 benchmarks), the scaling of the computational workload should be close to linear with an increase in workers. Although the experiments show significant speedup, that speedup is far from linear (Figure 3.1b). These results are identical to those in (48) and (19).

6.4 Batch Size Optimization

As both the computation and communication performance models depend on the (micro)batch configuration, I perform experiments with different numbers and sizes of batches

and micro-batches, as listed in Table 6.1. Note that "Config A" has been used for all previous experiments, because it leads to the lowest execution times without getting out of memory for any of the neural networks.

Table 6.1: Training time in seconds per epoch and computational scalability for different GPipe configurations with the ImageNet dataset using 4 GPUs.

Configuration	*Config A* *mb=24* *#mb=12* *b=288*	*Config B* *mb=32* *#mb=12* *b=384*	*Config C* *mb=32* *#mb=20* *b=640*	*Config D* *mb=12* *#mb=28* *b=336*	*Config E* *mb=16* *#mb=30* *b=480*
Equation 6.1	0.31	0.31	0.29	0.28	0.28
ResNet-18	1206.34	1030.49	936.27	2198.92	1664.14
ResNet-50	2839.55	2965.78	2374.64	3840.04	3190.78
ResNet-152	8742.24	7790.45	7053.19	10312.74	8087.80

The results show a complex pattern: using more micro-batches or larger micro-batch sizes does not always result in better performance. The benchmark with the largest total batch size (Config C) does have the lowest training time for all networks, but the second largest (Config E) is much slower than the third largest (Config B). Configurations with large micro-batches seem to perform better than those with a large number of micro-batches: as the scaling of the computational workload with the number of micro-batches (Equation 6.1) is an asymptotic function, increasing the number of micro-batches beyond some point no longer gives any significant performance improvement. When this number of micro-batches is reached, the micro-batch size should be increased instead as this does result in meaningful performance improvements. Reducing the number of micro-batches also reduces the number of communication steps in the forward and backward pass, which may result in better performance due to inefficiencies in torchgpipe's implementation (Figure 6.3).

6.5 Summary

The performance predictor for torchgpipe does not have the desired accuracy, as the predicted execution times deviate between 20 and 76 percent with those from the benchmark suite. Both at the computation and communication side torchgpipe behaves differently

than expected: on the compute side, all operations (e.g. additions, convolutions) are much less densely packed on the forward pass than on the backward pass. The backward pass is executed by PyTorch's backpropagation engine, while the forward pass is executed by torchgpipe itself, raising the suspicion that torchgpipe's implementation is far from efficient and so far from expected. On the communication side, torchgpipe does not overlap computation with communication, contrary to what is promised (19). Furthermore, the communication operations themselves takes much longer than expected. The performance predictor for torchgpipe could possibly be updated to reach a better prediction accuracy, however as torchgpipe does not perform as promised, I would first like to see torchgpipe updated.

7

PipeDream

PipeDream implements pipeline parallelism differently than GPipe: Model partitions can be trained on by multiple workers at a time (data parallelism), and there is no synchronization between workers after the processing of one batch - instead, batches are processed continuously (Figure 7.1). First, each worker performs a forward pass on zero or more warm-up batches so that the execution pipeline is filled. Hereafter, all workers alternate between forward and backward passes, on batch at a time, until all training data is processed. Compared to GPipe, this approach should result in a much better speedup of the training time with the number of workers, because the number of steps where all workers train in parallel depends on the number of batches for PipeDream, which tends to be much larger than the number of micro-batches used in GPipe (Chapter 6). Just like with GPipe, most (if not all) of the communication can be overlapped with computation once the pipeline is completely filled.

This is how PipeDream was originally designed in (20). However, there is one misconception by the authors in the implementation of the design: PyTorch's DistributedDataParallel (49), which is used to perform data parallelism within one model partition, does

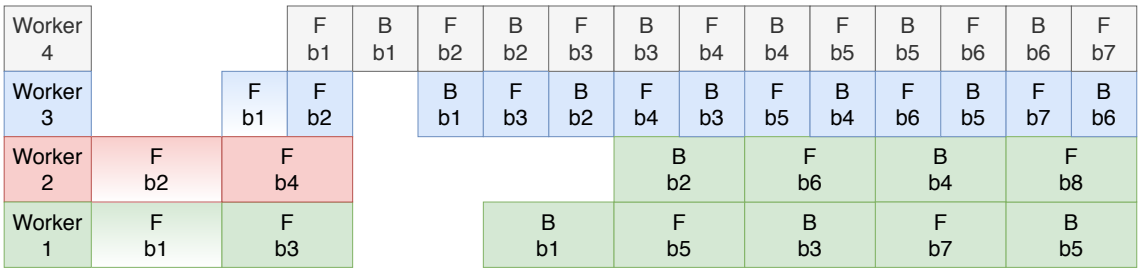


Figure 7.1: Predicted PipeDream execution pipeline with 4 workers in a 2-1-1 setup. Blocks in white represent warm-up batches.

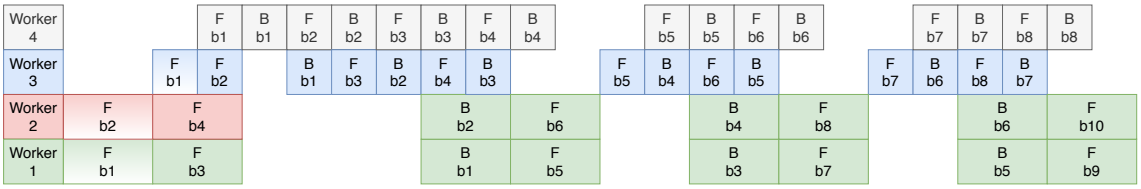


Figure 7.2: Actual PipeDream execution pipeline with 4 workers in a 2-1-1 setup. Blocks in white represent warm-up batches.

not support asynchronicity (wait-free backpropagation). This means that the interleaving of forward and backward passes by workers 1 and 2 in Figure 7.1 is not possible, instead PipeDream executes like in Figure 7.2. This has been confirmed by the analysis of PyTorch autograd and through manual instrumentation.

Moreover, as the difference in the number of workers per model partition grows (e.g. 3 workers in the first partition, 1 in the second partition), there will be less overlap between the workload of the two partitions, resulting in almost sequential execution. Therefore I have decided not to include PipeDream in the recommender system. Moreover, it is currently not possible to use large batch sizes in PipeDream without manual analysis and partitioning of neural networks, because the automatic profiler included in PipeDream performs analysis on a full neural network on 1 GPU only, which may not always be possible due to the limited memory capacity of a single GPU (Section 3.4.2).

7.1 Supporting PipeDream

As PipeDream is implemented as a standalone framework instead of a library which can be imported into your code, I have extended the source code of PipeDream to support the datasets and neural networks used in the benchmark suite. This includes support for the profiler and runtime system to load the datasets and neural networks. For the support of more neural networks, I extend PipeDream's intermediate representation to support more layer types.

Dividing the batches from a dataset over all workers can be quite complex in PipeDream, as each model partition can be executed using data parallelism. Figure 7.3a shows a problem with original implementation: if the number of batches in a dataset is not perfectly divisible by the number of workers in each model partition, PipeDream would crash. This is quite problematic, as either the amount of data or the configuration of the pipeline needs to be changed to make the application function again. I have built support for this situation by using the greatest common divisor (gcd): the number of batches in the dataset

is lowered until it can be perfectly be divided by the gcd of the number of workers in each model partition. This guarantees that all data that enters the pipeline will be processed by all workers (Figure 7.3b). In the future, this approach can be further improved on by using oversampling: Increasing the number of batches instead of decreasing may give better accuracy as training data is not dropped.

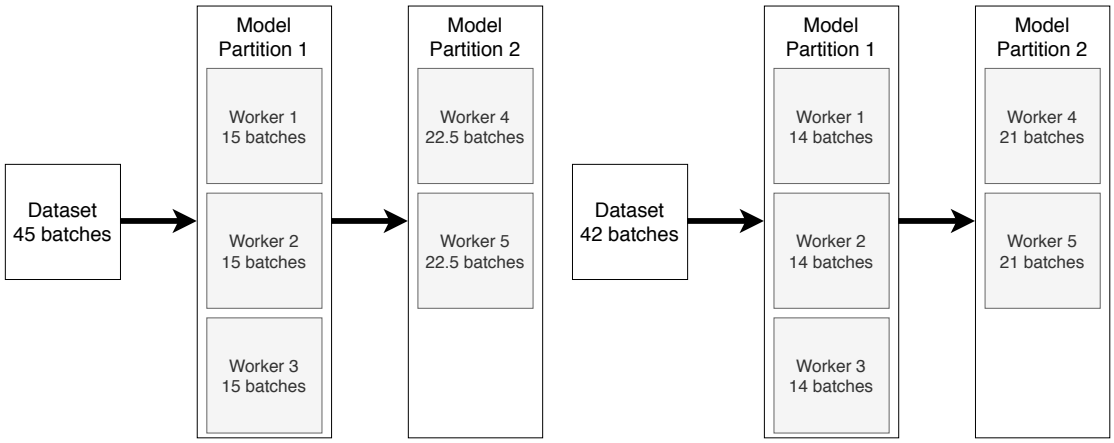


Figure 7.3: Division of batches over workers in PipeDream for a 3-2 configuration.

Additionally, PipeDream would crash when using a configuration where one model partition uses more workers in parallel than the previous model partition (e.g., in the case of a 1-3 configuration). Although I have fixed this particular problem, PipeDream can still deadlock during execution for unclear reasons, so more in-depth analysis is needed.

7.2 Summary

The implementation of PipeDream has several problems: While asynchronous data parallelism is promised (20), PipeDream does use the synchronous variant, which results in a major loss of performance. Furthermore, PipeDream has several bugs which hurt the usability of the framework, as PipeDream may crash or deadlock without notice. Although I have improved the existing framework by fixing most of the usability related problems, the performance of PipeDream is still worse than GPipe (Chapter 3) and much worse than promised (42). Because of this, I have chosen not to create a performance predictor for PipeDream.

8

Recommender System

The recommender system uses the performance predictions for Horovod and GPipe to advise the user which distribution model to use for a particular application. Specifically, the system will *recommend the distribution model that achieves the lowest training time*, after filtering out the models that cannot execute a particular application due to using more memory than available or other circumstances are ignored. In this chapter, we discuss the accuracy of our recommender system.

8.1 Recommendations

In Table 8.1 I compare the recommendations of the recommender system with the results from the benchmark suite to see how accurate the recommender system is. The first thing that stands out is that Horovod always is the fastest distribution model unless the memory usage exceeds the capacity of the GPUs, which is the case with the benchmark with ResNet-152 and Highres. There GPipe is correctly recommended as it uses significantly less memory per worker compared to Horovod.

Besides this one exception, for the ImageNet and Highres datasets which contain high resolutions images, Horovod is always correctly recommended to the user as with data parallelism the compute workload is more efficiently distributed over all workers. For MNIST and CIFAR-10, datasets containing relatively low resolution images, this differs as there communication overhead is more important due to the low amount of compute workload available. As the pipeline parallelism of GPipe should have less communication overhead than the data parallelism of Horovod (Chapter 2), it should often be the faster distribution model of the two. However as the implementation of GPipe is inefficient (Chapter 6) compared to what it promises to do (19), Horovod is currently still faster.

	MNIST	CIFAR-10	ImageNet	Highres
ResNet-18	GPipe Horovod 97.1 %	GPipe Horovod 103.0 %	Horovod Horovod -	Horovod Horovod -
ResNet-50	Horovod Horovod -	GPipe Horovod 56.0 %	Horovod Horovod -	Horovod Horovod -
ResNet-152	Horovod Horovod -	GPipe Horovod 46.1 %	Horovod Horovod -	GPipe GPipe -
VGG-11	GPipe Horovod 91.5 %	GPipe Horovod 76.7 %	Horovod Horovod -	Horovod Horovod -
VGG-16	GPipe Horovod 63.4 %	GPipe Horovod 62.7 %	Horovod Horovod -	Horovod Horovod -
MobileNet v2	Horovod Horovod -	Horovod Horovod -	Horovod Horovod -	Horovod Horovod -

Table 8.1: Comparison of the recommendations made by the recommender system with the results from the benchmark suite. Each cell shows the recommended distribution model, the actual fastest distribution model according to the benchmark suite and the relative difference in training time between these two respectively.

This also explains the large differences in training time between GPipe and Horovod for these mispredictions (Table 8.1).

8.2 Accuracy

The recommender system correctly recommends the fastest distribution model to the user for 66 percent of the benchmarks in the benchmark suite (Table 8.2). However the accuracy of the recommender system differs drastically between datasets: 50 percent for MNIST, 16 percent for CIFAR-10 and 100 percent for both ImageNet and Highres. As mentioned in Section 8.1, the inaccuracies for the MNIST and CIFAR-10 datasets are due to the relatively

large communication overhead coupled with the inefficient implementation of GPipe. As long as the inefficiency of GPipe (and PipeDream) remain, a better recommendation for a user compared to the current system would be to use Horovod as long as the memory usage does not exceed the memory capacity of the hardware, and use GPipe otherwise.

	MNIST	CIFAR-10	ImageNet	Highres
ResNet-18	F	F	T	T
ResNet-50	T	F	T	T
ResNet-152	T	F	T	T
VGG-11	F	F	T	T
VGG-16	F	F	T	T
MobileNet v2	T	T	T	T

Table 8.2: Correctness of the predictions of the recommender system on the applications in the benchmark suite for 4 GPUs. A correct prediction is marked with True (T), an incorrect prediction with False (F).

The recommender system has been calibrated using the results from the benchmark suite, that is the performance models have been updated until they could estimate the training time of the benchmarks accurately (although this still is not always the case, see Table 8.1). This approach works as the benchmark suite includes various datasets, neural networks and distribution models, so the calibration process results in a generalizable recommender system. As the current recommender system should be generalizable to some extend, users with DL applications that are not included in the benchmark suite do not have to reconfigure the performance models to get results with similar accuracy as reported in Table 8.2.

8.3 Alternatives

These results raise the question if there are alternatives for such a recommender system which achieve a higher prediction accuracy. The recommender system currently uses a mix of analytical and statistical methods to predict the training time of a benchmark using one of the available distribution models. As the performance predictor is inaccurate for some benchmarks with Horovod and for many benchmarks with GPipe, the resulting recommendations are inaccurate as well. The inaccuracy of the performance predictor is

on the one hand due to too high-level abstractions in the performance models, and on the other hand due to the distribution models not functioning as advertised.

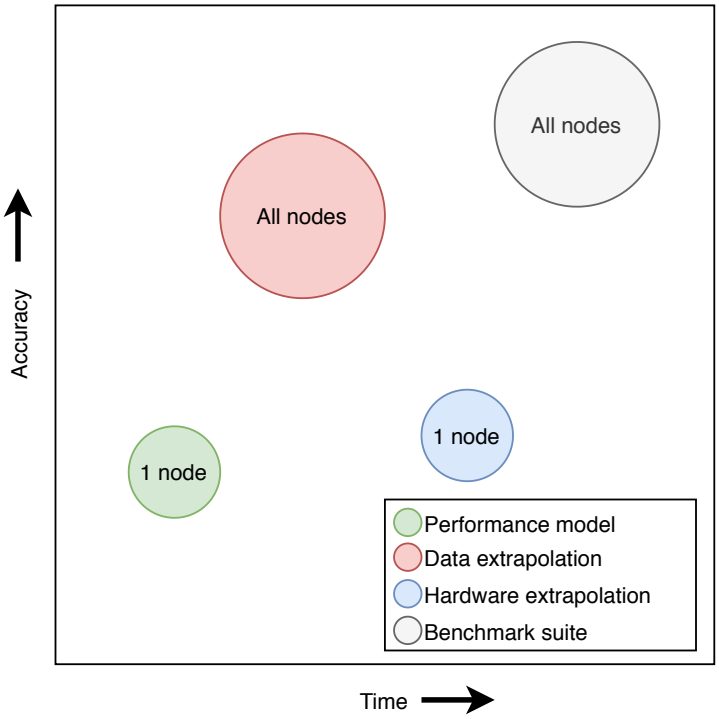


Figure 8.1: A comparison of four recommender systems in terms of time and resources needed by the recommender system and accuracy predictions.

I discuss three alternatives here. First, use the benchmark suite directly, as that will result in actual training times instead of predicted ones. However, running a benchmark suite takes a lot of time and resources, which is not desirable in most circumstances. The execution times of the benchmarks in the suite can be used to understand how the training time scales with different neural networks, datasets and distribution models. Second, use the same extrapolation method, as proposed in Chapter 4 for the PyTorch single GPU performance predictor on the distribution models. The recommender system would execute a benchmark on as many nodes as the user wants to predict for, and train the model on a couple of batches, whereafter the training-time per batch is extrapolated to the number of batches in the entire dataset. While this method, called data extrapolation, does take up little time as training is performed on only a few batches, it requires the use of as many nodes as the user wants to predict for. Third, use the extrapolation method on hardware instead of data. This method uses the training time of a sequential application, together with some assumptions on the scalability of the targeted distribution model, to estimate

the training time on multiple nodes. The limitations of this method are that it takes a lot of time to execute a sequential DL application and the neural network should fit on a single device, however compared to data extrapolation only a single hardware node is needed. In Figure 8.1 the trade-offs between the four methods is shown: getting more accurate recommendations compared to the analytical and statistical performance model method involves executing more parts of the application that you are trying to predict for which requires more time and resources.

9

Conclusion

The field of deep learning has seen great progress over the last couple of years due to the development of more expressive neural networks. Not only has the time to train these neural networks increased significantly, the billions of parameters these networks contain do not fit on a single device anymore, making distributed deep learning necessary. There exist multiple distribution models which either implement data, model or pipeline parallelism, each with their own advantages and disadvantages. Due to the complexity of these models, in-depth analysis of a deep learning application is needed before the optimal distribution model in terms of training time and accuracy for that application can be determined.

In this thesis, I have created a recommender system that, given a model of a deep neural network, a dataset, and a hardware configuration, recommends the fastest distribution model to the user. Using a newly created distributed deep learning benchmark suite with varying datasets, neural networks and distribution models, I calibrated the recommender system, resulting in a 100 percent prediction accuracy for datasets with high dimension images. However, during the process I discovered that both GPipe and PipeDream do not function as their design suggests, resulting in low prediction accuracies for datasets with small resolution images due to unexpected behaviour by these distribution models. This results in Horovod currently being the fastest distribution model when the memory capacity of the hardware is not exceeded, in which case GPipe should be used.

There are several limitations to the current version of the recommender system and improvements that can be made accordingly. First, the performance predictor, which drives the recommender system, struggles with accurately predicting communication overhead due to its dynamic nature. The analytical model used to predict the communication overhead could be expanded upon to more accurately take communication schemes and latency and throughput between GPUs into account. Second, I have focused on one particular GPU

model for all experiments, which may threaten the validity of the recommender system. In the future this can be improved upon by executing the benchmark suite and the recommender system on different types of hardware to verify if the performance predictors are generalizable across different hardware devices and so the prediction accuracy of the recommender system is equal across these devices. Third, I have only taken GPUs into account in favor of CPUs due to their superiority in compute power. However, CPUs boost multiple advantages which make them interesting for deep learning compared to GPUs such as increased memory capacity. The recommender system can be ported to support CPUs besides GPUs, however the main problem is that multiple distribution models only support GPUs at the moment, which limits the effectiveness of a recommender system. Finally, the implementations of torchgpipe and PipeDream do not deliver the expected performance which lowers the accuracy of the recommender system. This can be improved upon by updating the implementations of these two distribution models, otherwise their performance models in the recommender system have to be adapted to more accurately model the current behaviour of the distribution models.

This project can be extended in a couple of ways. For example, data and hardware extrapolation methods can be looked into to find out how the current extrapolation method used for the sequential predictor can be improved upon. What is the minimal training unit in terms of data and hardware upon which you can accurately extrapolate? Another extension could be to execute the recommender system on a CPU so the sequential deep learning predictor can handle large models, and then translate the training time on the CPU to that on a GPU. This requires a deep understanding of CPU and GPU hardware as a translation using maximum CPU and GPU FLOPS is most likely too shallow and will result in inaccurate predictions.

Appendix A

Neural Networks

The benchmark suite includes four datasets (MNIST, CIFAR-10, ImageNet and Highres) and six neural networks (ResNet-18, ResNet-50, ResNet-152, VGG-11, VGG-16 and MobileNet v2). As the datasets differ in the resolution of the images and the number of classes the images are divided in, the neural networks need to be adjusted for each dataset.

I use the neural networks included in PyTorch (8) for ImageNet as a basis so all versions of the same neural network function similarly. Highres shares its neural networks with ImageNet as they have the same number of classes and differ only a factor of two in image width and height. However as the Highres dataset uses larger image resolutions, the output size of each layer is larger, resulting in a larger activation and gradient memory.

Meanwhile MNIST and CIFAR-10 have separate models because of significant differences in image resolution and color profile. For CIFAR-10 the neural networks from (43) are used as there the models from PyTorch are already adapted to CIFAR-10. The detailed information of each neural network has been acquired using (50) and can be found in the tables below. The design of these is inspired by the original ResNet (16), VGG (17) and MobileNet (15) papers.

For the MobileNet v2 models in Tables A.4, A.7, A.10 and A.13 the following applies: Each line describes a sequence of 1 or more identical (modulo stride) layers, repeated n times. The first layer of each sequence has stride s, all others use a stride of 1. All spatial convolutions use a 3 3 kernel. The expansion factor t is used in Table A.1. To account for the smaller images in the MNIST and CIFAR-10 data sets compared to ImageNet, the stride of some layers has been lowered.

Table A.1: Bottleneck residual block for MobileNet v2. Transforms k to $k0$ channels, with stride s and expansion factor t.

Input			*Operator*		*Output*		
h	w	k	1	1 conv2d, ReLU6	h	w	(tk)
h	w	(tk)	3	3 dwise s=s, ReLU6	$\frac{h}{s}$	$\frac{w}{s}$	(tk)
$\frac{h}{s}$	$\frac{w}{s}$	(tk)	linear 1	1 conv2d	$\frac{h}{s}$	$\frac{w}{s}$	$k0$

A.1 MNIST

Table A.2: ResNet for MNIST.

Layer name	*Output size*		*18-layer*	*50-layer*	*152-layer*
conv1_x	28	28	$\begin{bmatrix} 3 & 3,64 \end{bmatrix}$ 1		
			$\begin{bmatrix} 3 & 3,64 \\ 3 & 3,64 \end{bmatrix}$ 2	$\begin{bmatrix} 1 & 1,64 \\ 3 & 3,64 \\ 1 & 1,256 \end{bmatrix}$ 3	$\begin{bmatrix} 1 & 1,64 \\ 3 & 3,64 \\ 1 & 1,256 \end{bmatrix}$ 3
conv2_x	14	14	$\begin{bmatrix} 3 & 3,128 \\ 3 & 3,128 \end{bmatrix}$ 2	$\begin{bmatrix} 1 & 1,128 \\ 3 & 3,128 \\ 1 & 1,512 \end{bmatrix}$ 4	$\begin{bmatrix} 1 & 1,128 \\ 3 & 3,128 \\ 1 & 1,512 \end{bmatrix}$ 8
conv3_x	7	7	$\begin{bmatrix} 3 & 3,256 \\ 3 & 3,256 \end{bmatrix}$ 2	$\begin{bmatrix} 1 & 1,256 \\ 3 & 3,256 \\ 1 & 1,1024 \end{bmatrix}$ 6	$\begin{bmatrix} 1 & 1,256 \\ 3 & 3,256 \\ 1 & 1,1024 \end{bmatrix}$ 36
conv4_x	4	4	$\begin{bmatrix} 3 & 3,512 \\ 3 & 3,512 \end{bmatrix}$ 2	$\begin{bmatrix} 1 & 1,512 \\ 3 & 3,512 \\ 1 & 1,2048 \end{bmatrix}$ 3	$\begin{bmatrix} 1 & 1,512 \\ 3 & 3,512 \\ 1 & 1,2048 \end{bmatrix}$ 3
	1	1	average pool, 10-d fc		

Table A.3: VGG for MNIST.

Layer name	*Output size*		*11-layer*	*16-layer*
conv1_x	14	14	$\begin{bmatrix} 3 & 3, 64 \end{bmatrix}$ 1	$\begin{bmatrix} 3 & 3, 64 \end{bmatrix}$ 2
			2 2 max pool, stride 2	
conv2_x	7	7	$\begin{bmatrix} 3 & 3, 128 \end{bmatrix}$ 1	$\begin{bmatrix} 3 & 3, 128 \end{bmatrix}$ 2
			2 2 max pool, stride 2	
conv3_x	3	3	$\begin{bmatrix} 3 & 3, 256 \end{bmatrix}$ 2	$\begin{bmatrix} 3 & 3, 256 \end{bmatrix}$ 3
			2 2 max pool, stride 2	
conv4_x	1	1	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 2	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 3
			2 2 max pool, stride 2	
conv5_x	1	1	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 2	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 3
	1	1	average pool, 10-d fc	

Table A.4: MobileNet v2 for MNIST.

Input size			*Operator*	t	n	s
28	28	1	conv2d	-	1	1
28	28	32	bottleneck	1	1	1
28	28	16	bottleneck	6	2	1
28	28	24	bottleneck	6	3	2
14	14	32	bottleneck	6	4	2
7	7	64	bottleneck	6	3	1
7	7	96	bottleneck	6	3	2
4	4	160	bottleneck	6	1	1
4	4	320	conv2d 1 1	-	1	1
4	4	1280	avgpool 4 4	-	1	-
1	1	1280	10-d fc	-	-	-

A.2 CIFAR-10

Table A.5: ResNet for CIFAR-10.

Layer name	*Output size*		*18-layer*	*50-layer*	*152-layer*
				$\begin{bmatrix} 3 & 3,64 \end{bmatrix}$ 1	
conv1_x	32	32	$\begin{bmatrix} 3 & 3,64 \\ 3 & 3,64 \end{bmatrix}$ 2	$\begin{bmatrix} 1 & 1,64 \\ 3 & 3,64 \\ 1 & 1,256 \end{bmatrix}$ 3	$\begin{bmatrix} 1 & 1,64 \\ 3 & 3,64 \\ 1 & 1,256 \end{bmatrix}$ 3
conv2_x	16	16	$\begin{bmatrix} 3 & 3,128 \\ 3 & 3,128 \end{bmatrix}$ 2	$\begin{bmatrix} 1 & 1,128 \\ 3 & 3,128 \\ 1 & 1,512 \end{bmatrix}$ 4	$\begin{bmatrix} 1 & 1,128 \\ 3 & 3,128 \\ 1 & 1,512 \end{bmatrix}$ 8
conv3_x	8	8	$\begin{bmatrix} 3 & 3,256 \\ 3 & 3,256 \end{bmatrix}$ 2	$\begin{bmatrix} 1 & 1,256 \\ 3 & 3,256 \\ 1 & 1,1024 \end{bmatrix}$ 6	$\begin{bmatrix} 1 & 1,256 \\ 3 & 3,256 \\ 1 & 1,1024 \end{bmatrix}$ 36
conv4_x	4	4	$\begin{bmatrix} 3 & 3,512 \\ 3 & 3,512 \end{bmatrix}$ 2	$\begin{bmatrix} 1 & 1,512 \\ 3 & 3,512 \\ 1 & 1,2048 \end{bmatrix}$ 3	$\begin{bmatrix} 1 & 1,512 \\ 3 & 3,512 \\ 1 & 1,2048 \end{bmatrix}$ 3
	1	1	average pool, 10-d fc		

Table A.6: VGG for CIFAR-10.

Layer name	*Output size*		*11-layer*	*16-layer*
conv1_x	16	16	$\begin{bmatrix} 3 & 3, 64 \end{bmatrix}$ 1	$\begin{bmatrix} 3 & 3, 64 \end{bmatrix}$ 2
			2 2 max pool, stride 2	
conv2_x	8	8	$\begin{bmatrix} 3 & 3, 128 \end{bmatrix}$ 1	$\begin{bmatrix} 3 & 3, 128 \end{bmatrix}$ 2
			2 2 max pool, stride 2	
conv3_x	4	4	$\begin{bmatrix} 3 & 3, 256 \end{bmatrix}$ 2	$\begin{bmatrix} 3 & 3, 256 \end{bmatrix}$ 3
			2 2 max pool, stride 2	
conv4_x	2	2	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 2	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 3
			2 2 max pool, stride 2	
conv5_x	1	1	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 2	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 3
			2 2 max pool, stride 2	
	1	1	average pool, 10-d fc	

Table A.7: MobileNet v2 for CIFAR-10.

Input size			*Operator*	t	n	s
32	32	3	conv2d	-	1	1
32	32	32	bottleneck	1	1	1
32	32	16	bottleneck	6	2	1
32	32	24	bottleneck	6	3	2
16	16	32	bottleneck	6	4	2
8	8	64	bottleneck	6	3	1
8	8	96	bottleneck	6	3	2
4	4	160	bottleneck	6	1	1
4	4	320	conv2d 1 1	-	1	1
4	4	1280	avgpool 4 4	-	1	-
1	1	1280	10-d fc	-	-	-

A.3 ImageNet

Table A.8: ResNet for ImageNet.

Layer name	*Output size*		*18-layer*	*50-layer*	*152-layer*
conv1	112	112	$7 \quad 7$, 64, stride 2		
conv2_x	56	56	$3 \quad 3$ max pool, stride 2		
			$\begin{bmatrix} 3 & 3,64 \\ 3 & 3,64 \end{bmatrix} 2$	$\begin{bmatrix} 1 & 1,64 \\ 3 & 3,64 \\ 1 & 1,256 \end{bmatrix} 3$	$\begin{bmatrix} 1 & 1,64 \\ 3 & 3,64 \\ 1 & 1,256 \end{bmatrix} 3$
conv3_x	28	28	$\begin{bmatrix} 3 & 3,128 \\ 3 & 3,128 \end{bmatrix} 2$	$\begin{bmatrix} 1 & 1,128 \\ 3 & 3,128 \\ 1 & 1,512 \end{bmatrix} 4$	$\begin{bmatrix} 1 & 1,128 \\ 3 & 3,128 \\ 1 & 1,512 \end{bmatrix} 8$
conv4_x	14	14	$\begin{bmatrix} 3 & 3,256 \\ 3 & 3,256 \end{bmatrix} 2$	$\begin{bmatrix} 1 & 1,256 \\ 3 & 3,256 \\ 1 & 1,1024 \end{bmatrix} 6$	$\begin{bmatrix} 1 & 1,256 \\ 3 & 3,256 \\ 1 & 1,1024 \end{bmatrix} 36$
conv5_x	7	7	$\begin{bmatrix} 3 & 3,512 \\ 3 & 3,512 \end{bmatrix} 2$	$\begin{bmatrix} 1 & 1,512 \\ 3 & 3,512 \\ 1 & 1,2048 \end{bmatrix} 3$	$\begin{bmatrix} 1 & 1,512 \\ 3 & 3,512 \\ 1 & 1,2048 \end{bmatrix} 3$
	1	1	average pool, 1000-d fc		

Table A.9: VGG for ImageNet.

Layer name	*Output size*		*11-layer*	*16-layer*
conv1_x	112	112	$\begin{bmatrix} 3 & 3, 64 \end{bmatrix}$ 1	$\begin{bmatrix} 3 & 3, 64 \end{bmatrix}$ 2
			2 2 max pool, stride 2	
conv2_x	56	56	$\begin{bmatrix} 3 & 3, 128 \end{bmatrix}$ 1	$\begin{bmatrix} 3 & 3, 128 \end{bmatrix}$ 2
			2 2 max pool, stride 2	
conv3_x	28	28	$\begin{bmatrix} 3 & 3, 256 \end{bmatrix}$ 2	$\begin{bmatrix} 3 & 3, 256 \end{bmatrix}$ 3
			2 2 max pool, stride 2	
conv4_x	14	14	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 2	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 3
			2 2 max pool, stride 2	
conv5_x	7	7	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 2	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 3
			2 2 max pool, stride 2	
	1	1	average pool, 4096-d fc, 4096-d fc, 1000-d fc	

Table A.10: MobileNet v2 for ImageNet.

Input size			*Operator*	t	n	s
224	224	3	conv2d	-	1	2
112	112	32	bottleneck	1	1	1
112	112	16	bottleneck	6	2	2
56	56	24	bottleneck	6	3	2
28	28	32	bottleneck	6	4	2
14	14	64	bottleneck	6	3	1
14	14	96	bottleneck	6	3	2
7	7	160	bottleneck	6	1	1
7	7	320	conv2d 1 1	-	1	1
7	7	1280	avgpool 7 7	-	1	-
1	1	1280	1000-d fc	-	-	-

A.4 Highres

Table A.11: ResNet for Highres.

Layer name	*Output size*		*18-layer*	*50-layer*	*152-layer*
conv1	256	256	7 7, 64, stride 2		
conv2_x	128	128	3 3 max pool, stride 2		
			$\begin{bmatrix} 3 & 3,64 \\ 3 & 3,64 \end{bmatrix}$ 2	$\begin{bmatrix} 1 & 1,64 \\ 3 & 3,64 \\ 1 & 1,256 \end{bmatrix}$ 3	$\begin{bmatrix} 1 & 1,64 \\ 3 & 3,64 \\ 1 & 1,256 \end{bmatrix}$ 3
conv3_x	64	64	$\begin{bmatrix} 3 & 3,128 \\ 3 & 3,128 \end{bmatrix}$ 2	$\begin{bmatrix} 1 & 1,128 \\ 3 & 3,128 \\ 1 & 1,512 \end{bmatrix}$ 4	$\begin{bmatrix} 1 & 1,128 \\ 3 & 3,128 \\ 1 & 1,512 \end{bmatrix}$ 8
conv4_x	32	32	$\begin{bmatrix} 3 & 3,256 \\ 3 & 3,256 \end{bmatrix}$ 2	$\begin{bmatrix} 1 & 1,256 \\ 3 & 3,256 \\ 1 & 1,1024 \end{bmatrix}$ 6	$\begin{bmatrix} 1 & 1,256 \\ 3 & 3,256 \\ 1 & 1,1024 \end{bmatrix}$ 36
conv5_x	16	16	$\begin{bmatrix} 3 & 3,512 \\ 3 & 3,512 \end{bmatrix}$ 2	$\begin{bmatrix} 1 & 1,512 \\ 3 & 3,512 \\ 1 & 1,2048 \end{bmatrix}$ 3	$\begin{bmatrix} 1 & 1,512 \\ 3 & 3,512 \\ 1 & 1,2048 \end{bmatrix}$ 3
	1	1	average pool, 1000-d fc		

Table A.12: VGG for Highres.

Layer name	*Output size*		*11-layer*	*16-layer*
conv1_x	256	256	$\begin{bmatrix} 3 & 3, 64 \end{bmatrix}$ 1	$\begin{bmatrix} 3 & 3, 64 \end{bmatrix}$ 2
			2 2 max pool, stride 2	
conv2_x	128	128	$\begin{bmatrix} 3 & 3, 128 \end{bmatrix}$ 1	$\begin{bmatrix} 3 & 3, 128 \end{bmatrix}$ 2
			2 2 max pool, stride 2	
conv3_x	64	64	$\begin{bmatrix} 3 & 3, 256 \end{bmatrix}$ 2	$\begin{bmatrix} 3 & 3, 256 \end{bmatrix}$ 3
			2 2 max pool, stride 2	
conv4_x	32	32	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 2	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 3
			2 2 max pool, stride 2	
conv5_x	16	16	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 2	$\begin{bmatrix} 3 & 3, 512 \end{bmatrix}$ 3
			2 2 max pool, stride 2	
	1	1	average pool, 4096-d fc, 4096-d fc, 1000-d fc	

Table A.13: MobileNet v2 for Highres.

Input size			*Operator*	t	n	s
512	512	3	conv2d	-	1	2
256	256	32	bottleneck	1	1	1
256	256	16	bottleneck	6	2	2
128	128	24	bottleneck	6	3	2
64	64	32	bottleneck	6	4	2
32	32	64	bottleneck	6	3	1
32	32	96	bottleneck	6	3	2
16	16	160	bottleneck	6	1	1
16	16	320	conv2d 1 1	-	1	1
16	16	1280	avgpool 7 7	-	1	-
1	1	1280	1000-d fc	-	-	-

References

[1] Yanping Huang, Youlong Cheng, Ankur Bapna, et al. Gpipe: E□cient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019. iii, 2, 10, 12, 19, 34

[2] Fei-Yue Wang, Jun Jason Zhang, Xinhu Zheng, et al. Where does AlphaGo go: From church-turing thesis to AlphaGo thesis and beyond. *IEEE/CAA Journal of Automatica Sinica*, 3(2):113–120, 2016. 1

[3] Bobby Chesney and Danielle Citron. Deep fakes: A looming challenge for privacy, democracy, and national security. *Calif. L. Rev.*, 107:1753, 2019. 1

[4] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. *arXiv preprint arXiv:1901.06032*, 2019. 1

[5] Linhao Dong, Shuang Xu, and Bo Xu. Speech-transformer: a no-recurrence sequence-to-sequence model for speech recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5884–5888. IEEE, 2018. 1

[6] Martín Abadi, Ashish Agarwal, Paul Barham, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org. 1, 11, 13

[7] François Chollet et al. Keras. https://keras.io, 2015. 1, 11

[8] Adam Paszke, Sam Gross, Soumith Chintala, et al. Automatic di□erentiation in PyTorch. 2017. 1, 11, 13, 51

[9] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 99–104. IEEE, 2016. 1

[10] Priya Goyal, Piotr Dollár, Ross Girshick, et al. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017. 1

[11] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, et al. Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds. *CoRR*, abs/1903.12650, 2019. 1

[12] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010 [cited 2016-01-14 14:24:11]. 2, 15

[13] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009. 2, 15

[14] Olga Russakovsky, Jia Deng, Hao Su, et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. 2, 15

[15] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation. *CoRR*, abs/1801.04381, 2018. 2, 16, 51

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *CoRR*, abs/1512.03385, 2015. 2, 16, 51

[17] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 2, 16, 51

[18] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018. 2, 25, 27

[19] Chiheon Kim, Heungsub Lee, Myungryong Jeong, et al. torchgpipe: On-the-fly Pipeline Parallelism for Training Giant Models. 2020. 2, 10, 15, 18, 19, 33, 34, 36, 38, 43

[20] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, et al. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019. 2, 10, 12, 39, 41

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org. 5

[22] Andrew Trask. *Grokking deep learning*. Manning Publications Co., 2019. 5, 35

[23] Karanbir Singh Chahal, Manraj Singh Grover, Kuntal Dey, and Rajiv Ratn Shah. A hitchhiker's guide on distributed training of deep neural networks. *Journal of Parallel and Distributed Computing*, 137:65–76, 2020. 8, 12

[24] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. E□cient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839*, 2018. 8

[25] Vishakh Hegde and Sheema Usmani. Parallel and distributed deep learning. In *Tech. report, Stanford University*. 2016. 8, 12

[26] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009. 8

[27] Salem Alqahtani and Murat Demirbas. Performance Analysis and Comparison of Distributed Machine Learning Systems. *arXiv preprint arXiv:1909.02061*, 2019. 9, 12, 21

[28] Tianqi Chen, Mu Li, Yutian Li, et al. MXNet: A Flexible and E□cient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, abs/1512.01274, 2015. 11

[29] Yangqing Jia, Evan Shelhamer, Jeff Donahue, et al. Ca□e: Convolutional Architecture for Fast Feature Embedding. *CoRR*, abs/1408.5093, 2014. 11

[30] Frank Seide and Amit Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135, 2016. 11

[31] Junjie Bai, Fang Lu, Ke Zhang, et al. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx, 2019. 12

[32] Yu Liu, Cheng Chen, Ru Zhang, et al. Enhancing the Interoperability between Deep Learning Frameworks by Model Conversion. Technical Report MSR-TR-2019-36, Microsoft, November 2019. 12

[33] Cody Coleman, Deepak Narayanan, Daniel Kang, et al. Dawnbench: An end-to-end deep learning benchmark and competition. *Training*, 100(101):102, 2017. 12

[34] Peter Mattson, Christine Cheng, Cody Coleman, et al. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019. 12

[35] Tal Ben-Nun, Maciej Besta, Simon Huber, et al. A modular benchmarking infrastructure for high-performance and reproducible deep learning. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 66–77. IEEE, 2019. 12

[36] Xiangru Lian, Ce Zhang, Huan Zhang, et al. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 5330–5340, 2017. 12, 14

[37] Jeffrey Dean, Greg Corrado, Rajat Monga, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012. 12

[38] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. Predicting the computational cost of deep learning models. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3873–3882. IEEE, 2018. 12

[39] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019. 12, 21

[40] Horace He. The State of Machine Learning Frameworks in 2019. https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/, 2019. 13

[41] Papers with Code. https://paperswithcode.com/task/image-classification. Accessed: 2020-04-01. 15

[42] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, et al. Pipedream: Fast and e□cient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018. 16, 29, 41

[43] Kuangliu. pytorch-cifar. https://github.com/kuangliu/pytorch-cifar, 2020. 17, 51

[44] Chih-Chieh Yang and Guojing Cong. Accelerating Data Loading in Deep Neural Network Training. *arXiv preprint arXiv:1910.01196*, 2019. 17

[45] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, pages 1731–1741, 2017. 18

[46] Shaohuai Shi, Qiang Wang, and Xiaowen Chu. Performance modeling and evaluation of distributed deep learning frameworks on gpus. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, pages 949–957. IEEE, 2018. 21

[47] NVIDIA. CUDA Samples. https://github.com/NVIDIA/cuda-samples/, 2019. 27

[48] Yanping Huang, Yonglong Cheng, Dehao Chen, et al. GPipe: E□-cient Training of Giant Neural Networks using Pipeline Parallelism. *CoRR*, abs/1811.06965, 2018. 36

[49] PyTorch DistributedDataParallel. https://pytorch.org/docs/stable/nn.html#torch.nn.parallel.DistributedDataParallel, 2020. 39

[50] Shubham Chandel. pytorch-summary. https://github.com/sksq96/pytorch-summary, 2020. 51