SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

January 2022, Vol. 65 112103:1–112103:17
https://doi.org/10.1007/s11432-020-3182-1

Reveal training performance mystery between TensorFlow and PyTorch in the single GPU environment

Hulin DAI[1], Xuan PENG[1], Xuanhua SHI[1*], Ligang HE[2], Qian XIONG[1] & Hai JIN[1]

[1]*National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan* 430074*, China;*
[2]*Department of Computer Science, University of Warwick, Coventry* CV4 7AL*, UK*

Received 12 September 2020/Revised 14 December 2020/Accepted 3 February 2021/Published online 16 December 2021

Abstract Deep learning has gained tremendous success in various fields while training deep neural networks (DNNs) is very compute-intensive, which results in numerous deep learning frameworks that aim to offer better usability and higher performance to deep learning practitioners. TensorFlow and PyTorch are the two most popular frameworks. TensorFlow is more promising within the industry context, while PyTorch is more appealing in academia. However, these two frameworks differ much owing to the opposite design philosophy: static vs dynamic computation graph. TensorFlow is regarded as being more performance-friendly as it has more opportunities to perform optimizations with the full view of the computation graph. However, there are also claims that PyTorch is faster than TensorFlow sometimes, which confuses the end-users on the choice between them. In this paper, we carry out the analytical and experimental analysis to unravel the mystery of comparison in training speed on single-GPU between TensorFlow and PyTorch. To ensure that our investigation is as comprehensive as possible, we carefully select seven popular neural networks, which cover computer vision, speech recognition, and natural language processing (NLP). The contributions of this work are two-fold. First, we conduct the detailed benchmarking experiments on TensorFlow and PyTorch and analyze the reasons for their performance difference. This work provides the guidance for the end-users to choose between these two frameworks. Second, we identify some key factors that affect the performance, which can direct the end-users to write their models more efficiently.

Keywords deep learning, performance, comparison, TensorFlow, PyTorch

Citation Dai H L, Peng X, Shi X H, et al. Reveal training performance mystery between TensorFlow and PyTorch in the single GPU environment. Sci China Inf Sci, 2022, 65(1): 112103, https://doi.org/10.1007/s11432-020-3182-1

1 Introduction

Deep learning has gained tremendous success in many fields including computer vision, speech recognition, and natural language processing (NLP). It relies on training deep neural networks (DNNs). The networks have been becoming deeper and more complex, and revolutionize from convolution neural network (CNN) and recurrent neural network (RNN), to Transformer and graph neural network (GNN). Training needs enormous computation power, which requires lots of hardware to accelerate it, such as GPU, TPU [1], and FPGA. Among these hardware choices, the GPU is the prevalent choice currently. The complexity from both applications and the continuous evolution of DNNs and hardware motivates the developers of deep learning frameworks to provide better usability and higher performance to deep learning practitioners.

There are many deep learning frameworks nowadays, including Caffe [2] developed by UC Berkeley, TensorFlow [3] by Google, PyTorch [4] by Facebook, CNTK [5] by Microsoft, MxNet. Currently, TensorFlow and PyTorch are two most popular frameworks. TensorFlow is appealing thanks to its vibrant community and good visualization, while PyTorch is popular because of its easy programming and debugging. TensorFlow and PyTorch differ much owing to their opposite design philosophies: static vs. dynamic computation graph. TensorFlow follows data as code and code is data idiom, which builds a

* Corresponding author (email: xhshi@hust.edu.cn)

computation graph before it starts running. Programming in PyTorch is more dynamic: the users can define and execute graph nodes as the execution goes along. Therefore, TensorFlow is capable of performing graph optimizations to have more efficient execution, while PyTorch could suffer from the overhead of Python interpreter and lack the opportunity of holistic optimizations compared to TensorFlow. Nonetheless, there are some voices claiming that PyTorch could be faster than TensorFlow [6][1)2)3)], which should not be ignored. There are many studies [6–13][4)] which evaluate various deep learning frameworks on different hardware. However, they usually focus on the output results of the frameworks to compare these deep learning frameworks, but not to explain the underlying reasons systematically for the output. The reason why there is little discussion about the underlying reasons is that it is very hard to compare these two frameworks owing to their totally different software stack, including the way to construct a computation graph, runtime scheduling.

In this work, we aim to dig deep into TensorFlow and PyTorch and to reveal the mystery of their performance comparison in single-GPU training. To make this analysis as comprehensive as possible, we select seven very popular neural networks, which includes CNN, RNN, and Transformer and also covers computer vision, speech recognition, and natural language processing. Note that this work focuses on single GPU training, because multi-GPU or multi-node training will introduce communication time and parameters aggregation time, and there are a lot of studies [14–18] that focus on the optimizations of multi-GPU and multi-node training. We believe the multi-GPU and multi-node training is deserving another piece of in-depth work. Nonetheless, we present some multi-GPU training speed results in Section 5 and prove that our findings can still hold in distributed training. Our major findings are summarized below.

• Most of the model training time is consumed by GPU computation time, which indicates that the implementation of key kernels plays a vital role in GPU training, such as convolution layer, long short-term memory (LSTM) [19] layer.

• The convolution layer can have many different implementations with different speeds. Faster algorithms usually need much more memory. TensorFlow has a better memory management than PyTorch, which means that when the model is big or the training batch size is large, TensorFlow is a better choice than PyTorch.

• The best convolution algorithm needs to profile the available algorithms at runtime. The users have three options for choosing the algorithm, including attempting all available algorithms for the best one (with extra profiling overhead), using a heuristic approach for obtaining a suitable algorithm (suboptimal performance), or using the default convolution algorithm (poor performance). For the first option, when the input size is fixed, this process only happens at the first mini-batch, which has little influence on the overall training. However, when the input size is varied, this process continues through the first epoch, which introduces non-trivial overhead, but delivers better performance in the next epoch. So when users want to try a new network (by running a few steps), it is better to use the cuDNN's heuristic approach for obtaining the best algorithm. But if users have decided the networks (run lots of epochs), this profiling should be enabled.

• There are a few LSTM implementations on GPU currently, e.g., LSTMCell which is the most basic implementation, LSTMBlockCell[5)], and cuDNNLSTM[6)]. cuDNNLSTM is the fastest and can be 10× faster than the basic one [20]. However, it is very popular to regularize RNNs, such as weight decay, dropout, and zoneout [21], which preserves the information instead of dropping hidden units in dropout. Some of them can only be built with more basic LSTM implementation (like LSTMCell), which are not available in well-optimized cuDNNLSTM. So, on the one hand, practitioners should use more well-optimized LSTM implementation when they can. On the other hand, the kernel writers should provide more flexible `APIs` while achieving high performance.

• Graph optimizations show little impact on the training speed in most neural networks and should not be considered for the purpose of training speed when hesitating between TensorFlow and PyTorch.

1) Tensorflow issues 21881. https://github.com/tensorflow/tensorflow/issues/21881.

2) Reddit discussion: why is Tensorflow so slow. https://www.reddit.com/r/MachineLearning/comments/8iguaw/d_why_is_tensorflow_so_slow/.

3) Reddit discussion: Why is PyTorch as fast as (and sometimes faster than) TensorFlow? https://www.reddit.com/r/MachineLearning/comments/cvcbu6/d_why_is_pytorch_as_fast_as_and_sometimes_faster/.

4) Baidu. DeepBench: benchmarking deep learning operations on different hardware. 2017. https://github.com/baidu-research/DeepBench

5) Lstmblockcell of tensorflow. http://man.hubwiz.com/docset/TensorFlow.docset/Contents/Resources/Documents/api_docs/python/tf/contrib/rnn/LSTMBlockCell.html.

6) Optimizing recurrent neural networks in cuDNN 5. https://devblogs.nvidia.com/optimizing-recurrent-neural-networks-cudnn-5/, 2016.

2 Background

2.1 DNN model and deep learning training

The emergence of DNN has enabled the rapid development of deep learning, and has received extensive attention in the artificial intelligence (AI) communities. Starting from AlexNet [22], various DNN architectures (such as GoogLeNet [23], ResNet [24]) appear in a short time and provide better feature detection and accuracy. Generally, the DNN structures are composed of an input layer, an output layer and multiple hidden layers in between. The layers can be regarded as a set of operations. Some frameworks such as Caffe use layer abstractions, while others such as Caffe2[7)] and TensorFlow use operation (or operator) abstraction. For different applications and purposes, there are different types of layers, such as convolution layer, pooling layer and activation layer for feature extraction in image classification, attention layer for filtering information in NLP, and the LSTM layer for saving memory. Even considering the existing layers, the combination of layers can also be explored to meet the application requirements, such as ResNet.

Deep learning training aims to find a suitable set of model parameters to minimize the loss function, which reflects the error between the prediction result of the sample and the ground truth label. The training process typically consists of millions of iterations, each of which involves two computationally intensive phases, i.e., forward and backward propagation. In the forward propagation, the training samples are fed into the input layer, together with weights and bias to calculate the output feature map, which is used as the input of the next layer. Finally, the forward propagation ends with the calculation of loss by comparing the output with the ground truth label at the output layer. The backward propagation (more commonly called backpropagation [25]) starts from the output layer, traverses the layers in reverse, calculates the gradients of parameters in each layer according to the loss value through the chain rule, and optimizes the parameters. There are many optimizers for backpropagation, such as stochastic gradient descent (SGD), Momentum [26] and Adam [27]. Generally, as the number of iterations increases, the loss value becomes smaller and smaller. Training will end when a certain condition is reached, for example, the loss value is less than a threshold, or the accuracy of the validation is higher than a threshold.

2.2 Deep learning frameworks

Deep learning frameworks can be divided into two categories: declarative and imperative frameworks. TensorFlow[8)], Caffe, and CNTK embrace the declarative programming. PyTorch adopts the imperative model. MxNet [28] uses the programming model mixed with these two. TensorFlow announces the imperative programming, eager execution [29], as its default execution mode from version 2.0. However, it is a new feature in TensorFlow, which is still in rapid development. So we only study its stable declarative programming in this work. This limitation is also explained in Section 5. Among these frameworks, TensorFlow and PyTorch are the two most popular frameworks in industry and academia. Therefore, we focus on the TensorFlow and PyTorch in this work.

TensorFlow. TensorFlow is a framework for machine learning which is developed by Google. It uses the data flow graph to describe computation, where the graph nodes represent the mathematical operations and the edges represent the data (Tensor in TensorFlow). The users use the `API` in TensorFlow to build the models, which will be transformed into an internal computation graph at TensorFlow runtime. In this process, TensorFlow can perform some optimizations to the graph for improving the performance. For example, pruning and constant folding can be used to simplify the graph. Common subexpression elimination (CSE) can be applied to eliminate computations. Owing to the characteristic of static computation graph in TensorFlow, it is much more difficult to handle RNN because it usually needs a dynamic computation graph.

PyTorch. PyTorch is a framework revolutionized from Torch [30] and developed by Facebook. It also follows the dataflow paradigm, and is same as TensorFlow. However, it offers more flexible `APIs`, which can define dynamic computation graphs. It embraces the "pythonic" coding style, which makes it simple to learn and use by the users. Tensor is also the core data abstraction in PyTorch which represents the multi-dimensional arrays. To facilitate the users, both PyTorch and TensorFlow offer automatic differentiation (also known as backpropagation) for all operations on Tensors. The difference is that:

7) Caffe2 framework, 2017. https://caffe2.ai.

8) Although TensorFlow supports these two programming models currently, it adopts the declarative programming model first, which lasts for a long time.

TensorFlow builds the graph of backpropagation before running, while PyTorch tracks the operations and builds the graph at runtime.

3 Experimental methods

3.1 Workloads selection

To make our analysis as comprehensive as possible, the testing workloads in this work cover the major application domains and also include the mainstream architectures of current DNNs.

3.1.1 *Computer vision*

Computer vision is a field of artificial intelligence that trains the computers to interpret and understand the visual world. It includes the applications, such as image classification, object detection. Since AlexNet beats the traditional methods and shows deep learning is a promising method for image classification, the neural networks in this field have been developing rapidly. In this work, we choose three most representative models: VGG16 [31], ResNet-50 [24], and InceptionV3 [32]. All these three models are CNN.

3.1.2 *Speech recognition*

DeepSpeech2 [33] is an end-to-end automatic speech recognition engine from Baidu, which is constructed with the convolution layer, the RNN layer, and the fully connected (FC) layer. It can be used to recognize either English or Chinese, two very different languages. Tacotron2 [34] is a neural network architecture for speech synthesis directly from text. It is composed of two components: a sequence-to-sequence network followed by a modified WaveNet vocoder. The major layers in Tacotron2 are the convolution layer, LSTM layer, and FC layer.

3.1.3 *Natural language processing*

Natural language processing is a branch of artificial intelligence that enables machines to read, understand, and derive meaning from human language. RNN is the pioneer of the networks that are applied in deep learning because the dependency in the language sequence is suitable for RNN to process. In recent years, the attention mechanism is becoming popular in NLP. Based on that, Google first abandons traditional CNN and RNN networks, but uses attention to build the entire network, which is called Transformer [35]. Therefore, we select two popular networks: GNMT [36], which is an LSTM-based network with attention, and BERT [37] from Google, which is a transformer-based model and breaks the records of many NLP tasks at the time when it is released.

3.2 Unify the implementations between TensorFlow and PyTorch

The implementations of the same neural network could vary between TensorFlow and PyTorch in a few aspects, which impacts the training performance and affects a fair comparison. Therefore, we try to unify the implementations of TensorFlow and PyTorch so as to provide a fair comparison. We present the implementation from two aspects: model structure and hyper-parameters. The key model settings are listed in Table 1 [24, 31–34, 36–39].

3.2.1 *Model structure*

In general, the structure of a model is fixed when its corresponding paper is published. However, the detailed implementations in workloads on different frameworks could vary. In our experiment, only the implementation of Tacotron2 differs between TensorFlow and PyTorch while others are the same. For Tacotron2, the implementation in TensorFlow is aligned with the model structure in the paper, while PyTorch's implementation is modified in the LSTM regularization, which uses the normal dropout to replace the original zoneout[9]. The decoder in PyTorch has an attention RNN layer, which is implemented by an LSTM layer, an attention layer and a decoder RNN layer, which is implemented by an LSTM layer,

9) Tacotron2 implemented in PyTorch. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/SpeechSynthesis/Tacotron2.

while in TensorFlow's implementation, a decoder RNN is implemented by two LSTM layers followed by an attention layer. In order to unify them, we replace zoneout in TensorFlow with dropout, and change the decoder in PyTorch's implementation to be the same as that in TensorFlow's. In addition, we only focus on the sequence-to-sequence network of Tacotron2 as it is the main architecture of Tacotron2 while others are the plug-in networks.

3.2.2 *Hyper parameters*

The hyper-parameters are configurable parameters with the default value, such as learning rate, dropout rate and batch size. We set the same hyper-parameter values in TensorFlow and PyTorch. Note the batch size will influence the training speed greatly. We list the default batch size of our experiments in Table 1. In the experiments, the results are obtained with the default batch size unless stated otherwise.

3.3 Obtaining accurate training speed

Training DNNs can last from weeks to months. To gather the training performance efficiently, we only sample a short run of the entire training owing to the iteration characteristic of deep learning training. However, the sampling periods could vary from the input. For the models with the fixed-length input, such as CNNs, the training could stabilize very soon which means the difference between iterations is very small. Therefore, we can gather an accurate training performance with a short period of running. For the models with the input of variable length, such as RNNs, the training speed of each iteration is different owing to different input sizes. In this case, it is necessary to train an epoch (traverse the entire dataset) to obtain the stable performance.

Besides, there is usually a warm-up phase at the beginning of the training, which is used to construct the computation graph, allocate memory, and explore some parameters (i.e., the workspace size of different convolution layers). Only after that, the computation of each step would show a repeated behavior, which can be used to represent accurate performance.

Next, we will describe the methodology of obtaining the accurate training speed in these seven models.

3.3.1 *CNNs*

We set 10 warmup steps to ensure that the training speed is stable enough. Then we record the elapsed time of 200 steps after warmup to calculate the average running time of a step. So, the training speed (images/s) can be calculated by the following formula:

$$\text{speed} = \frac{\text{batch\_size} \times \text{num\_steps}}{\text{elapsed\_time}}.$$

3.3.2 *RNNs & BERT*

We use steps/s to express the speed in RNN and BERT. Because profiling the convolution algorithm could exist during the entire first epoch in DeepSpeech2 and Tacotron2 (explained in Subsection 4.4.1), which affects the training performance, we use the run-time of the second epoch to calculate the speed. We run 200 and 300 steps and calculate the average speed in GNMT and BERT, respectively.

4 Evaluation

4.1 Experimental setup

Our experiment is conducted on the GPU computing type gn5 in Alibaba Cloud[10)], which has an NVIDIA Tesla P100 GPU, 2.50 GHz Intel(R) Xeon(R) CPU E5-2682 v4 processors with 8 logical cores, 60 GB RAM and 440 GB local NVMe SSD, running Ubuntu 16.04 OS. The CUDA version is 10.0. cuDNN is 7.6.4. The CUDA driver is 418.87.01. Our models are trained and tested on TensorFlow-GPU v1.15.2 and PyTorch v1.4.0 using the data type of 32-bit floating-point.

10) Alibaba Cloud. https://www.aliyun.com/.

Table 1 The workloads and settings in TensorFlow and PyTorch

Domain	Model		Key layer's implementation	BS	Dataset	Github repository	
						TensorFlow	PyTorch
CV	ResNet-50 [24] VGG16 [31] InceptionV3 [32]		Conv(cuDNN)	64	ImageNet [38]	TensorFlow/Benchmarks[d)]	PyTorch/Examples[i)]
SR	Tacotron2 [34]	Encoder	cuDNNLSTM	32	LJSpeech[a)]	Rayhane-Mamah/Tacotron-2[e)]	NVIDIA/DeepLearningExamples[j)]
		Decoder	LSTMBlockCell				
	DeepSpeech2 [33]		Conv (cuDNN), cuDNNLSTM	20	LibriSpeech [39]	TensorFlow/Models[f)]	SeanNaren/deepspeech.pytorch[k)]
NLP	GNMT [36]	Encoder	cuDNNLSTM	128	WMT'16 EN-DE[b)]	NVIDIA/DeepLearningExamples[g)]	NVIDIA/DeepLearningExamples[l)]
		Decoder	cuDNNLSTM				
	BERT [37]		Embedding, Full-Connect	32	wikitext-103-raw[c)]	Google-research/BERT[h)]	Huggingface/Transformers[m)]

a) Ito K. The LJ speech dataset. 2017 https://keithito.com/LJ-Speech-Dataset/.

b) ACL 2016 first Conference on Machine Translation (wmt16), 2016. http://www.statmt.org/wmt16/it-translation-task.html.

c) Merity S. The WikiText long term dependency language modeling dataset. https://blog.einstein.ai/the-wikitext-long-term-dependency-language- modeling-dataset/.

d) https://github.com/tensorflow/benchmarks.git.

e) https://github.com/Rayhane-mamah/Tacotron-2.

f) https://github.com/tensorflow/models.git.

g) https://github.com/NVIDIA/DeepLearningExamples.git.

h) https://github.com/google-research/bert.git.

i) https://github.com/pytorch/examples.git.

j) https://github.com/NVIDIA/DeepLearningExamples.git.

k) https://github.com/SeanNaren/deepspeech.pytorch.

l) https://github.com/NVIDIA/DeepLearningExamples.git.

m) https://github.com/huggingface/transformers.

Table 2 Overall training speed on TensorFlow and PyTorch

Model	TensorFlow	PyTorch	Difference (%)
ResNet-50	**216.14** (images/s)	206.15 (images/s)	4.85
VGG16	**150.32** (images/s)	131.54 (images/s)	14.28
InceptionV3	**145.18** (images/s)	138.12 (images/s)	5.11
BERT	1.73 (steps/s)	**1.77** (steps/s)	2.31
GNMT	**2.085** (steps/s)	2.030 (steps/s)	2.71
DeepSpeech2	0.696 (steps/s)	**0.735** (steps/s)	5.60
Tacotron2	0.286 (steps/s)	**0.307** (steps/s)	7.34

4.2 Overall training performance comparison

In this subsection, we first look into the comparison of overall training performance between TensorFlow and PyTorch. The results are shown in Table 2.

As we can see from the results, the performance difference between TensorFlow and PyTorch is trivial, which is less than 8% in most models except 14.28% in VGG16. Generally, TensorFlow exhibits better training performance on CNN models, while PyTorch is better on BERT and RNN models (except for GNMT). In conclusion, the training performance between TensorFlow and PyTorch is very close, which is different from some previous work or statements in [6, 11, 40]. The following two main reasons can be attributed to this result. (1) Frameworks version. The versions of TensorFlow and PyTorch in our experiment are all very stable, e.g., TensorFlow v1.15.2 is the last version (until the time we conducted the experiments) that uses the graph execution as the default execution, which means that the well-known optimizations have been implemented (e.g., asynchronous execution and using the NCHW data format in GPU). (2) The model. The detailed implementation of the same model or some hyper-parameter values could vary in different frameworks. But we have unified the model structure and hyper-parameters (described in Subsection 3.2).

Even though the overall training performance of TensorFlow and PyTorch is similar, the performance gap increases in some cases (e.g., different batch sizes), which will be shown next.

We now dig deep into the training process of TensorFlow and PyTorch to find out the reasons behind the results. This part is organized as follows. First, we break down the training process on a single GPU and analyze the procedure further in Subsection 4.3. Then, Subsection 4.4 presents how different implementations of key layers in current models affect the training speed. Finally, we reveal the benefit of a principle difference between TensorFlow and PyTorch, i.e., graph optimization, in Subsection 4.5.

4.3 GPU processing dominates the training process

Deep learning training in a single GPU can be decomposed into reading data from disk, preprocessing in CPU, memory copy, and GPU processing (including kernel launch and kernel execution). The memory copy includes data transfer from CPU to GPU and GPU to CPU. The former is mostly regarding the input data transfer, such as image, word embedding, while the latter is mostly regarding the transfer of parameters, which is used to update the model weights. From experiments, we observe that most of the model training time is consumed by GPU processing. In the majority of models, the GPU processing is mainly about GPU kernel execution (over 85.91%) while the GPU kernel launch time increases in RNN (Tacotron2 has the longest kernel launch time). We discuss this point as follows.

The training process is asynchronous in TensorFlow and PyTorch, which means that reading data from disk, preprocessing in CPU and memory copy can all be overlapped with GPU computing. Firstly, we test the training speed using ramdisk (the CPU memory is used to simulate the disk) and the original disk. The difference between them is very small and can be neglected. So, the impact of the disk can be ignored in a single-GPU training. We also test the CPU usage in the training process, which is relatively low such that it has no effect on overall training performance.

Then, we collect the overall training time, GPU kernel execution time, and memory copy time using the NVIDIA visual profiler (nvprof). We calculate the ratio of memory copy time and GPU kernel execution time to the overall training time, respectively. For memory copy, the ratio is less than 3% and 1.42% on average in all models. Combing with its overlapping with GPU computing, the influence on the training is trivial. For GPU kernel execution, the result is shown in Table 3. We can see that the percentage is greater than 96% for CNN models no matter in TensorFlow or PyTorch. The value on BERT also exceeds

Table 3 Percentage of GPU computation time

Model	TensorFlow (%)	PyTorch (%)
ResNet-50	96.38	96.03
VGG16	99.64	99.34
InceptionV3	96.23	97.02
BERT	98.96	92.59
GNMT	85.91	96.55
DeepSpeech2	87.04	92.69
Tacotron2	54.31	66.62

92% because of that matrix multiplication dominates in the computation of transformer-based models, while matrix multiplication is well-optimized operation in GPU. Thus we will focus mainly on CNN and RNN models in the next. For GNMT and DeepSpeech2, the percentages are around 86% on TensorFlow, and lager than 92% on PyTorch. The smallest ratio appears on Tacotron2 for two frameworks, which is caused by the GPU kernel launch time. From the nvprof result, we observe that there are a lot of time intervals between the GPU kernel executions along the timeline of Tacotron2's decoder. This is because Tacotron2 will enumerate the input at the dimension of time steps and launch the kernel to process each of the time steps (76649 kernel launch in one step vs. 447 in VGG16). On the one hand, the data size of a single time step is very small, which makes the kernel so tiny that the computation is completed quickly. On the other hand, the number of these tiny kernels is enormous. These are the two reasons why the GPU kernel execution time is far from enough to overlap the kernel launch time. Further, we increase the default batch size (32) of Tacotron2 to 64. We find that the percentage of GPU computation time increases from 54.31% to 78.2% in TensorFlow, and from 66.62% to 85.81% in PyTorch.

Insight 1. In the single-GPU training, the training process is dominated by GPU processing (kernel launch+kernel execution+memory copy). Memory copy shows nearly no effect on the training performance. For CNNs and transformer-based models, the training time is mainly consumed by kernel execution. For RNNs, the kernel launch occupies a small part of training owing to the long kernel launch stage, which cannot be overlapped by kernel execution. Therefore, optimizing the kernel algorithm and reducing the execution time of important kernels can improve the training performance effectively. How to fuse more kernels and reduce the overhead of kernel launch is also very important for RNNs.

4.4 Kernel implementation has big impact

4.4.1 *Convolution layer*

The convolution layer is the key layer in the field of computer vision. It is also the most time-consuming in CNN owing to the time complexity of convolution. Therefore, the execution time of convolution layers plays a key part in the overall training time of CNN.

There are seven implementations of the convolution algorithms totally in cuDNN that can be categorized into three kinds: GEMM-based, FFT-based, and Winograd's minimal filtering algorithms [41]. The GEMM transforms the input into a large matrix and another matrix is generated with the elements of each filter. Then, the result can be got by the matrix multiplication of both matrices. There are two variants of the basic GEMM: GEMM-impl and GEMM-impl-precomp, where the transformation is performed on the fly by the kernel that computes the GEMM. There is a tiled version of FFT, where the inputs are split into smaller tiles, and a fused version Winograd, where the transformations of inputs, filters, and outputs are included in the kernel that computes the multiplication. There is not the best algorithm for all convolution layers because of two reasons: (1) different input sizes prefer different convolution algorithms; (2) some algorithms need extra memory space to store the auxiliary variables (faster convolution algorithms need more space generally). Hence the algorithm with the best performance may not be available owing to the limit of GPU memory. Because of this, both TensorFlow and PyTorch allow to select the fastest algorithm for a specific input size in all available convolution algorithms. PyTorch implements this by calling a specific cuDNN `API` (`cudnnFindConvolutionForwardAlgorithmEx`[11]), which attempts all available cuDNN algorithms for forwarding convolution layer using the GPU memory allocated by users, and outputs the performance metrics. TensorFlow achieves this through running all available convolution algorithms at runtime and selecting the fastest one among them. Moreover, both frameworks cache the

11) It is for the forward convolution algorithm. The backward convolution also has the similar `API`.

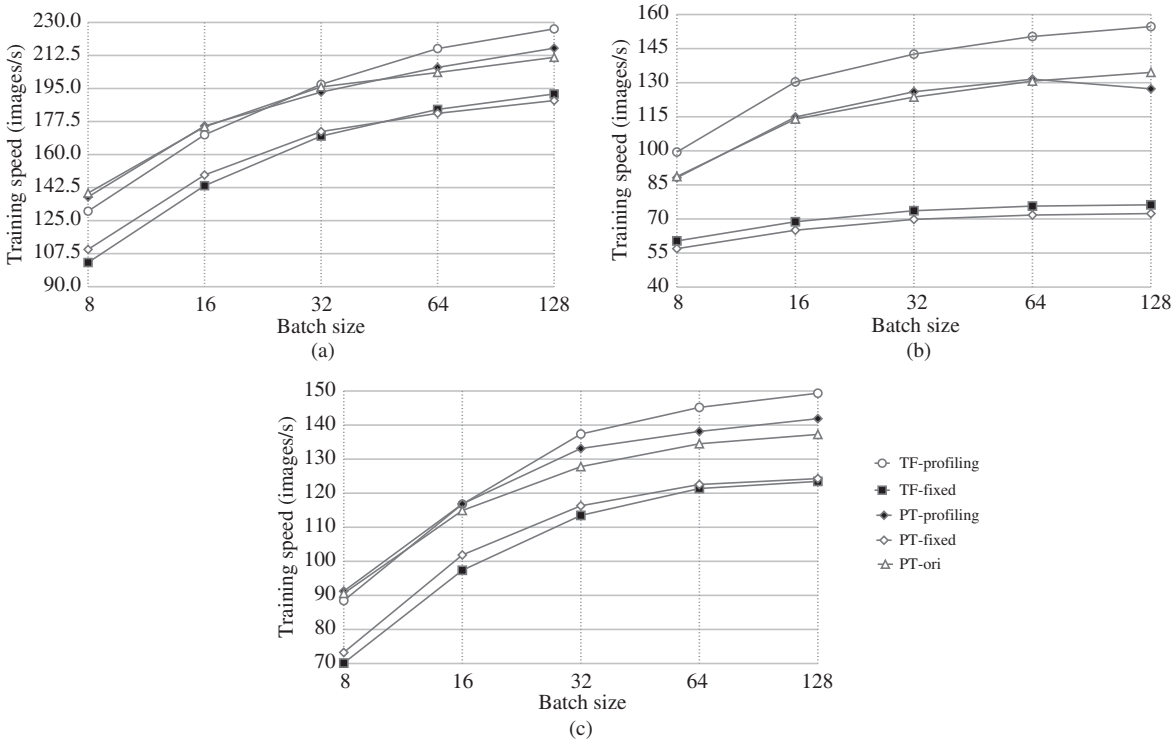

Figure 1 Training speed for (a) ResNet-50, (b) VGG16, (c) InceptionV3.

best convolution algorithm for this input size, and use the best one when meeting convolution of this input size again. This feature is denoted as profiling mode. Also, this feature can be disabled by the user in both TensorFlow and PyTorch through setting the environment variable and option, respectively. In this situation, the default convolution algorithm[12] will be chosen, which we call fixed mode. These two modes in TensorFlow and PyTorch are denoted by TF-profiling, TF-fixed, PT-profiling, and PT-fixed in the rest of the paper. The original TensorFlow also uses the profiling mode. In addition, PyTorch uses `cudnnGetConvolutionForwardAlgorithm_v7`[13] to select the algorithm by default, which uses a heuristic approach to evaluate the performance of all algorithms, and thus consumes a trivial amount of time. We denote this as PT-ori.

We first test the CNN models on these five settings to illustrate the effect of different convolution algorithms and compare the performance between TensorFlow and PyTorch. We also conduct the experiments with the batch size of 8 to 128 (8, 16, 32, 64, 128) to compare the results under different memory pressure. The results of ResNet-50, VGG16, and InceptionV3 are shown in Figure 1(a) and (b). For the convenience and clarity of comparison, we first look into the training speed between PT-ori and PT-profiling. They are nearly the same in most cases except for VGG16 with the batch size of 128 on which the speed of PT-ori is slightly faster than PT-profiling. This is because in the profiling mode, the cached best algorithm is determined only by the input size. It does not consider different convolution layers with the same input size. Their best algorithms could be different as the free GPU memory for the workspace is different for them. However, this difference is still small. So, we will mainly focus on the comparison of profiling mode and fixed mode in TensorFlow and PyTorch, which is listed below.

• **Profiling mode vs. fixed mode.** For TensorFlow, the training speed of profiling mode is 19.38% faster than the fixed mode for ResNet-50 on average. The value is 19.63% for InceptionV3. For VGG16, the profiling mode has the greatest performance improvement, which is over 102% with a batch size of 128. These results can be explained by the following two reasons: (1) the convolution layers account for more in VGG16 compared to ResNet-50 and InceptionV3; (2) the kernel sizes of convolution layers are big in VGG16, but small in ResNet-50 and InceptionV3, which means that the performance difference is more prominent between the convolution algorithms with the lowest and highest performance. In

12) The default convolution algorithms usually consume little memory but have low performance. They are coded in the source code of TensorFlow and PyTorch, and are the same in two frameworks.

13) The backward convolution also has the similar `API`.

Table 4 Training time of DeepSpeech2 and Tacotron2

Model	PT-ori (s)	PT-fixed (s)	PT-profiling (s)	
			1st epoch	2nd epoch
DeepSpeech2	2149.15	3081.88	2816.39	1940.25
Tacotron2	1412.98	1433.56	1520.39	1331.52

PyTorch, the result is similar.

• **TF-fixed vs. PT-fixed.** The training speeds of TF-fixed and PT-fixed are very close (the difference is 3.7% on average). This is because that the mainly GPU kernels are totally the same between TensorFlow and PyTorch, and thus show similar training performance. It also support our viewpoint: GPU processing dominates the training process.

• **TF-profiling vs. PT-profiling.** For the overall results, the difference in training speed between TensorFlow and PyTorch is within 6% in ResNet-50 and InceptionV3. On VGG16, TensorFlow is faster than PyTorch by 15% on average and by 21.5% maximum. Second, the performance advantage of TensorFlow becomes more prominent when increasing the batch size in all three models. On the one hand, this is because TensorFlow has a better memory management module than PyTorch owing to the fact that TensorFlow holds the whole computation information which can free unnecessary memory more timely. Therefore, when GPU memory pressure increases along with the batch size, TensorFlow can spare more available memory to opt for faster convolution algorithms compared to PyTorch (in one of the convolution layers in backpropagation of VGG16, PyTorch uses the default algorithms[14)], which consume 8.72 kB memory for filter algorithm and 75.63 kB for data algorithm, while TensorFlow uses faster algorithms[15)], which consume 3530 MB memory), and therefore, show a better training performance. On the other hand, recall that VGG16 has a larger convolution kernel size and hence its performance gap is the biggest among these three models.

Insight 2. In CNN models, the training performance is similar between TensorFlow and PyTorch. However, TensorFlow shows better training performance than PyTorch in larger batch size owing to its better GPU memory management.

We also test Tacotron2 and DeepSpeech2 which also have convolution layers. The biggest difference from the CNN models is that the input data size varies with iterations. Therefore, the profiling of convolution algorithms will occur during the first epoch, which adds non-trivial overhead. Hence, we show the performance of the first and second epoch separately in profiling mode. The result is shown in Table 4. Here, we only show the result of PyTorch as the original PyTorch uses a heuristic convolution algorithm choosing `API` that TensorFlow does not support. Besides, TensorFlow shows a similar result on the profiling and fixed mode in our tests.

We can see that in the first epoch, DeepSpeech2 consumes less training time in PT-ori than in profiling mode and fixed mode by 23.69% and 30.26%, respectively. For Tacotron2, the values are 7.06% and 1.44%. Owing to the power of profiling, which allows to choose the faster convolution algorithms, the profiling mode exhibits the best training speed in the second epoch. For DeepSpeech2, it takes less time than PT-ori and fixed mode by 9.72% and 37.04%, respectively. For Tacotron2, the difference narrows to 5.77% and 7.12%. The reason why Tacotron2 shows a smaller difference is mainly the dataset. When the input data size is different across iterations, the profiling overhead will occur in every iteration. When the input data size of subsequent iterations has appeared in a previous iteration, there is no profiling overhead, which amortizes the overall profiling overhead in the first epoch. Thus the performance difference will be smaller.

Insights 3. The convolution algorithm chosen in the models (usually RNN+CNN) with variable input length is a little tricky. Because the profiling of convolution algorithms, i.e., `cudnnFindConvolutionForwardAlgorithmEx`[11)], will introduce non-trivial overhead in the first epoch, but deliver a better performance in the following epoch. Therefore, when the end-users just want to try a neural network (run a few steps), using the heuristic convolution algorithm choosing `API` is the best choice, i.e., `cudnnGetConvolutionForwardAlgorithm_v7` in the original PyTorch. By the way, TensorFlow should also support this `API`. But when the end-users have determined the networks and need to train lots of epochs, the profiling mode will be the best option.

14) `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1` and `CUDNN_CONVOLUTION_BWD_DATA_ALGO_1` for backward filter and data algorithms.

15) `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_WINOGRAD_NONFUSED` and `CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD_NONFUSED` for backward filter and data algorithms.

Table 5 Various LSTM implementations on TensorFlow and PyTorch

Model		LC	LBC	cuDNNLSTM
DeepSpeech2		✔	✔	✔
GNMT		✔	✔	✔
Tacotron2	Encoder	✔	✔	✔
	Decoder	✔	✔	✗

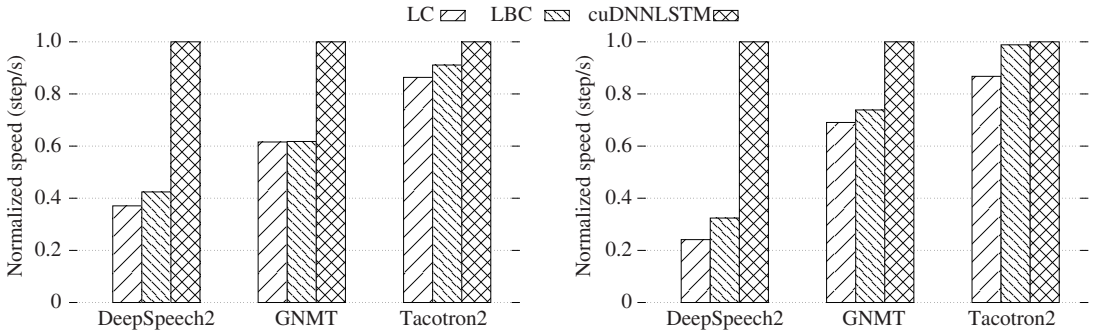

Figure 2 Normalized performance of different LSTM implementations in (a) TensorFlow and (b) PyTorch.

4.4.2 *LSTM layer*

LSTM neural networks, which stand for long short-term memory, are a type of recurrent neural networks. Simply put, LSTM networks have some internal contextual state cells that act as long-term or short-term memory cells. The output of the LSTM network is modulated by the state of these cells. This is a very important property when the prediction of the neural network needs to depend on the historical context of inputs, rather than only on the very last input. Although various LSTM implementations had been benchmarked [20], it only tests the basic LSTM layer which lacks analysis on the real-world neural networks. In addition, some of the statements in [20] are out-of-date, such as "TensorFlow cuDNNLSTM is not tested with variable length data as it does not support such input", which is already fixed in the new version of TensorFlow[16]. Therefore, we will test various LSTM implementations in the end-to-end models with state-of-the-art features.

There are mainly three GPU LSTM implementations currently, which are listed as follows. We evaluate these LSTM implementations on DeepSpeech2, GNMT, and Tacotron2. The availability of three LSTM implementations in these three models is shown in Table 5. In summary, except that cuDNNLSTM is not available in Tacotron2's decoder, all other implementations are supported. This is because the LSTM state of each time step needs to be processed separately in the decoder of Tacotron2 (irregular), while cuDNNLSTM can only perform the calculation on the data of all time steps (regular) currently (the latest version when we did the experiment).

• LSTMCell (LC): pure frameworks-based implementation, easy to modify. Loop over time is `tf.while_loop` in TensorFlow while Python `For` loop in PyTorch.

• LSTMBlockCell (LBC): optimized LSTM with a single operation per time-step. Loop over time is `tf.while_loop` in TensorFlow while Python `For` loop in PyTorch.

• cuDNNLSTM: cuDNN's LSTM implementation.

We evaluate various LSTM implementations on these three models, and show the training speeds in Figure 2(a) and (b). Note that the decoder of Tacotron2 cannot use cuDNNLST. Hence we use LBC for Tacotron2 decoder when testing cuDNNLSTM. For DeepSpeech2, using cuDNNLSTM is 2.69× faster than using LC on TensorFlow, and 4.14× faster on PyTorch. For GNMT, the speed difference is about 50% between cuDNNLSTM and LC on both TensorFlow and PyTorch. Various implementations of LSTM in Tacotron2's encoder have little effect on training speed. We can also see that the performance improvement of LBC is within 10% compared with LC, except for on DeepSpeech2, which is 34%.

Generally, cuDNNLSTM is the best-optimized implementation. Users should use it whenever they can. LBC shows little performance improvement compared to LC because it also needs to launch the kernel in each time step, which is the same as LC. Thus the kernel launch overhead is still the bottleneck of the performance.

16) Tensorflow pull requests 23588. https://github.com/tensorflow/tensorflow/pull/23588.

Insights 4. Among the LSTM implementations, cuDNNLSTM shows incomparable training performance. So end-users should use cuDNNLSTM in their models whenever possible. However, the flexibility of cuDNNLSTM is not compatible with its superior performance for now. For example, it does not offer an interface for end-users to perform some complicated LSTM regularization, e.g., zoneout. Hence, for the algorithm developers, it is necessary to continue improving the algorithm in both aspects of performance and flexibility.

4.5 Forget about graph optimization

A big difference between TensorFlow and PyTorch is graph optimization. Common knowledge is that declarative programming (e.g., TensorFlow) has better performance as it holds the entire computation graph and can perform lots of optimizations to the graph. However, there are also voices claiming that PyTorch has achieved better training speed than TensorFlow [6]. These two different views confuse practitioners about how to choose between TensorFlow and PyTorch from the perspective of training speed. Therefore, we investigate the effect of graph optimization on the training speed through a series of experiments.

First, we will give a brief introduction to the graph optimization in TensorFlow and what it does. There are 14 kinds of graph optimizations totally in current TensorFlow[17]. However, some of them are not enabled by default, such as autoparallel optimization. Some can be done when there is no computation graph, such as layout optimization, which is also available in PyTorch. So, we only care about the optimizations which can only be performed when there is a computation graph before running, which is listed below[18].

• **Pruning optimization.** Pruning the nodes that have no effect on the output from the graph. It is usually run first to reduce the size of the graph and speed up the processing in other optimizations.

• **Constant folding.** Inferring the value of tensors statically when possible by folding constant nodes in the graph and materializing the result using constants.

• **Arithmetic optimization.** Simplifying the arithmetic operations by eliminating common subexpressions and simplifying arithmetic statements.

• **Dependency optimization.** Removing or rearranging control dependencies to shorten the critical path for a model step or enable other optimizations; also removing the nodes that are effectively no-Ops such as identity.

In the following, the graph optimizations are referred to the above four optimizations.

4.5.1 *Overall performance without graph optimizations*

First, we evaluate the training speed of the seven models by disabling the graph optimizations. The result is shown in Figure 3. We can see that when disabling graph optimizations, the performance degradation is less than 0.6% for CNNs; for GNMT and DeepSpeech2 the degradation is less than 2%; the value is about 2.5% for BERT. The biggest performance loss is on Tacotron2, which is close to 15%.

Generally, we can see that the performance degradation by disabling graph optimizations is negligible in most models except for Tacotron2. Next, we will find out what happens in the graph optimizations and explain why the performance loss is larger in Tacotron2.

4.5.2 *Digging into graph optimizations*

The graph optimizations essentially reduce the number of computation nodes in a graph. For example, pruning removes unnecessary nodes. Constant folding uses less constant nodes to replace constant nodes in the graph. The arithmetic operation removes some common computation nodes. Therefore, we use the number of reduced nodes to represent the effect of graph optimizations. The result is shown in Figure 4. Note that constant folding will add some nodes (constant node), and therefore the sum could exceed 100%. We organize the results of graph optimizations according to the model architecture.

• CNN: graph optimizations reduce about 55% of the nodes, in which pruning reduces about 45% nodes and arithmetic operation reduces roughly 5%.

• RNN: graph optimizations reduce over 70% of the nodes, in which pruning reduces over 50% and arithmetic and dependency together reduce about 20%.

17) Specifically for TensorFlow v1.15.2.

18) TensorFlow graph optimization with Grappler. https://www.tensorflow.org/guide/graph_optimization.

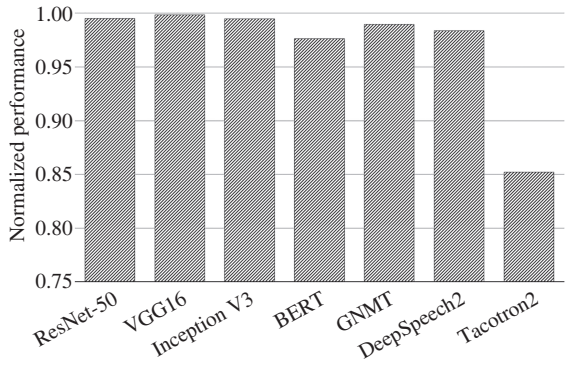

Figure 3 Normalized performance without graph optimizations.

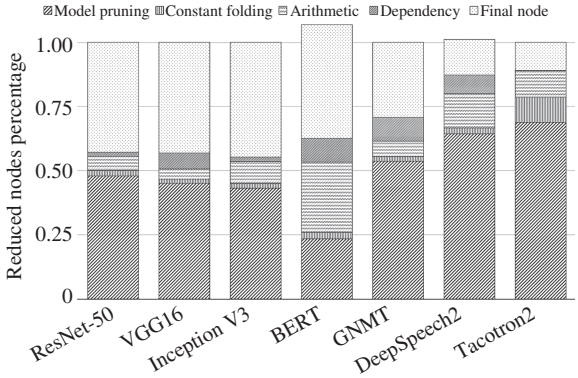

Figure 4 Reduced nodes percentage of each graph optimization.

Table 6 Percentage of operations' types

Model	Expensive Ops (%)	Cheap Ops (%)	No Ops (%)
ResNet-50	0.0397	9.4766	90.4837
VGG16	0.173	11.9377	87.8893
InceptionV3	0.0255	10.4699	89.5046
BERT	0	8.7593	91.2407
GNMT	0.6309	12.8987	86.4704
DeepSpeech2	0.1421	13.6461	86.2118
Tacotron2	0.5241	20.6355	78.8404

• CNN+RNN: graph optimizations reduce over 85% of the nodes. Especially the value nearly reaches 90% for Tacotron2. The pruning reduces about 65% of the nodes.

• Transformer: graph optimizations reduce about 60% of the nodes, in which pruning reduces about 23% and arithmetic reduces about 27%.

Further, we classify these reduced nodes into three categories according to the calculation time of their corresponding operations, which is listed below.

• Expensive operations: operations such as convolution calculation, matrix multiplication, fast fourier transformation and RNN, which have complicated numerical calculations.

• Cheap operations: operations such as add, sub, multiplication, division, whose computation is very fast on GPU.

• No operations: operations such as Const, Assign, Identity, Shape, which only perform value assignment or variable initialization and do not perform any numerical calculation.

Then we count the types of the operations corresponding to the reduced nodes and calculate the percentage. The result is shown in Table 6. The majority of the operations are no Ops which accounts for over 90%, whose computation time can be totally ignored. As for the expensive Ops, they are less than 1% in all models, and thus introduce trivial overhead to the overall runtime. Therefore, the graph optimizations deliver little performance improvement in most cases.

For Tacotron2, we profile one step of its execution by enabling and disabling graph optimizations and count the execution time of those reduced nodes. The execution time with/without graph optimizations is 3.402820 s and 3.896510 s, respectively, in which the difference is 0.49369 s. The total execution time of those reduced nodes is 0.508513 s, which is very close to the time difference. This result means this difference is caused by the execution overhead of extra nodes, not by the scheduling overhead. This also supports our first viewpoint: GPU time dominates the training process. We also found that most of these reduced nodes are created at the pre-processing phase, which does not affect the core computation of the models. Therefore, it does not introduce any problems in PyTorch, which is consistent with the competitive performance of PyTorch in Tacotron2.

Insights 5. Graph optimization in TensorFlow shows trivial training performance speedup in most models. This is because the graph optimization mainly reduces the nodes whose operations are very cheap (even without any numerical computations) to compute or some initialization nodes. Therefore, the advantages of graph optimization should not be considered from the perspective of training speed.

5 Limitation of this work and discussion

This work does not cover the multi-GPU and multi-node training as they need to introduce different factors, such as inter-GPU bandwidth, inter-node bandwidth, that makes the training performance of a specific model be tightly related to GPU models, communication bandwidth (including inter-device and inter-node), parameter synchronization strategy, even CPU type [14, 42], which warrants another piece of in-depth work. Although we will not make any conclusion in multi-GPU or multi-node environments in the scope of this paper, we intend to prove that the insights presented in this paper still hold in the distributed training scenario. Therefore, we benchmark the CNN models, an RNN model (GNMT), and a transformer-based model (BERT) in multi-GPU environment and show the overall training performance, which is listed in Table 7. We discuss the performance of distributed training in the following.

This work does not study the final training accuracy, inference speed of different deep learning frameworks as it falls into another scope of deep learning. This work does not include some new features such as PyTorch's TorchScript[19] and TensorFlow's eager execution [29]. The former aims to improve the performance and deployment through serialization in PyTorch while the latter enables imperative programming in TensorFlow. These two features intend to bridge the gap between TensorFlow and PyTorch. But the models need to be rewritten. Only a few models support these two features currently. So, we leave these for future work. This work does not investigate the compilers in deep learning, e.g., TensorFlow XLA[20], TVM [43] and PyTorch GLOW [44]. Except for the official deep learning compiling tools for each framework, there are a lot of research studies aiming to accelerate the deep learning procedure via compiling optimizations. Recently, Rammer [45] optimizes the execution of DNN models by holistically exploiting parallelism through inter- and intra- operator co-scheduling. TASO [46] proposes an automatic graph substitutions approach to optimize the data-flow graph. AutoTVM [47], FlexTensor [48], Tensor comprehension [49], and Halide [50] try to tune and generate efficient hardware-specific operator code. Therefore, we leave the comparison of the compiling techniques between TensorFlow and PyTorch for future work.

Distributed training discussion. There are two different methods to run distributed training, i.e., data parallelism and model parallelism, in which data parallelism is the most popular approach as model parallelism needs careful model partition and operator placement that is non-trivial and usually costly [51]. Hence, we mainly focus on data parallelism. The experiment environment has two NUMA CPUs connected via QPI. There are 4 NVIDIA V100 GPUs (32 GB GPU memory with NVlink) split into two groups and 4 GPUs are connected as a ring. Except for the bi-directional bandwidth between the second GPU and third GPU is $\approx$ 50 GB/s (1 NVLink connection), the left bi-directional bandwidth between GPUs is $\approx$ 100 GB/s (2 NVLinks connection). The CPU and GPUs are connected by PCI-e 3.0×16 (16 GB/s theoretical bandwidth). The software environment is consistent with Section 4.

Need to note that the synchronization strategy needs to be unified in TensorFlow and PyTorch, and then the training performance comparison is meaningful. There are two synchronization strategies in data parallelism, i.e., all-reduce ana Parameter Server (PS). All-reduce aggregates every GPU's gradients in a collective manner before GPUs update their own parameters locally, while PS architecture consists of workers and PS, in which workers push the gradients to PS and PS aggregates the gradients from different workers and updates the parameters. In PyTorch, there are two `API` to support distributed training, i.e., `torch.nn.parallel.DistributedDataParallel` and `torch.nn.DataParallel`. They all parallelize the model by splitting the input across the specified devices by chunking in the batch dimension, and replicate the model on each device to handle a portion of the input. During the backward propagation, gradients from each replica are summed into the original model. The difference is that `torch.nn.DataParallel` is single-process multi-thread parallelism and naturally suffers from Python GIL contentions while `torch.nn.parallel.DistributedDataParallel` uses multi-process parallelism and hence there is no GIL contention across model replicas. We use the latter one (all-reduce) to run distributed training as it is recommended by PyTorch official document. In TensorFlow, both PS and all-reduce are supported in distributed training but with considerable code modification compared to PyTorch. Here, CNN models use all-reduce but GNMT's implementation in TensorFlow only supports PS mode. The official BERT's implementation in TensorFlow does not support multi-GPU, so we use a third-party's implementation[21] which leverages Horovod [52] with NCCL as backend. For all experiment

19) TORCHSCRIPT. https://pytorch.org/docs/stable/jit.html.
20) XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla.
21) BERT's multi-GPU mplementation with Horovod. https://github.com/lambdal/bert.

Table 7 Multi-GPUs training speed and communication overhead

		GPU number	Model				
			ResNet-50 (images/s)	VGG16 (images/s)	InceptionV3 (images/s)	GNMT (steps/s)	BERT (steps/s)
Training speed	TensorFlow	1	368.78	243.60	240.38	3.29	3.241
		2	723.42	474.35	475.87	3.122	3.117
		4	1440.03	963.91	956.92	2.731	3.031
	PyTorch	1	344.918	225.813	236.533	3.125	3.08
		2	657.027	431.794	–	2.89	2.91
		4	1316.561	838.252	–	2.882	2.89
Communication overhead	TensorFlow	2	1.917%	2.638%	1.017%	5.106%	3.838%
		4	2.379%	1.077%	0.478%	16.991%	6.467%
	PyTorch	2	4.756%	3.608%	–	7.520%	5.519%
		4	4.574%	6.737%	–	7.782%	6.169%

in multi-GPU, the batch size in Table 1 is per GPU batch size which means the global batch size needs to multiply by the number of GPUs.

Firstly, we can see that the training performance comparison between TensorFlow and PyTorch in a single V100 GPU is basically aligned with the conclusion (BERT in TensorFlow is a bit faster than PyTorch, but the difference is with 5%) in P100 GPU from Table 7. In a multi-GPU environment, for CNN models, InceptionV3's multi-GPU training speed is blank as its multi-GPU implementation in PyTorch has bugs. The communication overhead is calculated by the following formula, in which m_step/s is the step/s of multi-GPU training speed and s_step/s is the step/s of single-GPU:

$$\text{communication overhead} = 1 - \frac{\text{m\_step/s}}{\text{s\_step/s}}.$$

It is obvious that the communication overheads in CNN models are all very low which means the models are scaling well with more GPUs. The largest is 6.737% that is PyTorch's VGG16 running on 4 GPUs. It is because that VGG16 is the largest model in these three CNN models that makes the parameter synchronization take more time compared to the other two models. For GNMT and BERT, they are also scaling well using NCCL as backend whose communication overheads are less than 8%. The largest communication overhead happens in GNMT of TensorFlow on 4 GPUs using PS mode. It results from that PS mode transfers parameters and gradients using PCI-e lane between CPU and GPU which is far lower than high bandwidth NVLinks between GPUs leveraged by NCCL. Thus the backward computation is not enough to overlap the data transfer and parameter update.

In summary, we can see that when TensorFlow and PyTorch adopt the same parameter synchronization strategy (all-reduce in this benchmark), the communication overheads are all very low across all models in the benchmark. Furthermore, the communication overheads of the same models in TensorFlow and PyTorch are also very close. That means the model computation part (i.e., the computation in single-GPU) becomes the major different part in multi-GPU and multi-node environment. Then the discussion falls back to the scope of this paper: training performance comparison between TensorFlow and PyTorch in single-GPU environment. Therefore, the insights presented in this paper could still hold in a distributed training scenario.

6 Related work

There is much research that focuses on benchmarking the deep learning workloads, but varies in the target and granularity of benchmarking. In this section, we discuss the existing research according to its benchmarking target.

Benchmark deep learning frameworks. There are two kinds of granularity to benchmark deep learning frameworks: model and operation. The former tests the performance metrics of the entire model while the latter tries to analyze it from the lower-level (such as matrix multiplication). DeepBench from Baidu measures the number of operations that appear in models. In such low-level profiling, it is hard to capture the characteristics of the entire neural network. Ai Matrix [9] evaluates at both two granularities. TBD [8] focuses on the profiling training on GPUs and measures the characteristics, e.g., CPU and GPU utilization and memory consumption. However, it only summarizes the experiment results and does not explain the reasons behind the results. Thus it cannot provide general advice to end-users. MLPerf [7] and

some other studies [6, 10–12, 53] benchmarked not only the throughput but also accuracy, while we focus on the system-level metrics but leave the domain-specific metric (accuracy) for application practitioners to explore. Specifically, Ref. [6] carried out their experiments in the CPU environment. Besides, Ref. [54] tried to model the performance in distributed training on GPUs.

Benchmark heterogeneous hardware. Ref. [55] was a recent work that proposed a benchmark methodology for analyzing deep learning and conducted the experiments on Intel CPU, Nvidia GPU, and Google TPU. Ref. [56] evaluated deep learning tools in docker containers, while the work in [57] was deployed on HPC architectures. Ref. [58] presented the performance analysis on both CPU- and GPU-based deep learning training. Ref. [13] performed the evaluation on GPUs and FPGA, and showed their characteristics. Ref. [59] focused on selecting the optimum models and frameworks for inference at edge devices. Our work only tests the GPU at the moment as GPU is still the most prevalent and well-developed deep learning training platform. We plan to extend our work to more hardware platforms, such as TPU.

7 Conclusion

The ultimate goal of this paper is to help the end-users make an informed decision about how to choose between the two most popular deep learning frameworks: TensorFlow and PyTorch, in the single-GPU training. We evaluate the single-GPU training on TensorFlow and PyTorch systematically using seven representative models. Through these comprehensive experiments, we provide insightful observations and advice to both end-users and system developers. Firstly, we break down the single-GPU training process to show that the training process is mainly consumed by GPU processing, which is mostly the kernel execution time. Therefore, the running speed of the key layers plays a vital role in single-GPU training. Then, we evaluate the performance of various models using different implementations of key layers and present the trade-off between them, in order to shed light to the end-users about choosing various implementations in reality. Finally, we evaluate the effect of the main difference between TensorFlow and PyTorch, i.e., graph optimization, on the training performance. The conclusion is that it should not be considered when making decisions between TensorFlow and PyTorch from the angle of performance.

Acknowledgements This work was partly supported by National Key R&D Program of China (Grant No. 2017YFC0803700) and National Natural Science Foundation of China (Grant No. 61772218). We thank Fan YANG, Ying CAO and Xiaosong MA for their valuable comments.

References

1 Jouppi N P, Young C, Patil N, et al. In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, Toronto, 2017. 1–12
2 Jia Y, Shelhamer E, Donahue J, et al. Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM International Conference on Multimedia, 2014. 675–678
3 Abadi M, Barham P, Chen J, et al. TensorFlow: a system for large-scale machine learning. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, 2016. 265–283
4 Paszke A, Gross S, Chintala S, et al. Automatic differentiation in PyTorch. In: Proceedings of the Autodiff Workshop on NIPS, 2017
5 Seide F, Agarwal A. CNTK: Microsoft's open-source deep-learning toolkit. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016. 2135–2135
6 Shi S, Wang Q, Xu P, et al. Benchmarking state-of-the-art deep learning software tools. In: Proceedings of the 7th International Conference on Cloud Computing and Big Data (CCBD), 2016. 99–104
7 Mattson P, Cheng C, Coleman C, et al. MLPerf training benchmark. In: Proceedings of the 3rd Conference on Systems and Machine Learning (SysML'20), 2020
8 Zhu H, Akrout M, Zheng B, et al. TBD: benchmarking and analyzing deep neural network training. In: Proceedings of International Symposium on Workload Characterization (IISWC 2018), 2018
9 Zhang W, Wei W, Xu L, et al. AI matrix: a deep learning benchmark for Alibaba data centers. 2019. ArXiv:1909.10562
10 Liu L, Wu Y, Wei W, et al. Benchmarking deep learning frameworks: design considerations, metrics and beyond. In: Proceedings of 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018. 1258–1269
11 Shatnawi A, Al-Bdour G, Al-Qurran R, et al. A comparative study of open source deep learning frameworks. In: Proceedings of the 9th International Conference on Information and Communication Systems (ICICS), 2018. 72–77
12 Coleman C, Narayanan D, Kang D, et al. DAWNBench: an end-to-end deep learning benchmark and competition. Training, 2017, 100: 102
13 Karki A, Keshava C P, Shivakumar S M, et al. Detailed characterization of deep neural networks on GPUs and FPGAs. In: Proceedings of the 12th Workshop on General Purpose Processing Using GPUs, 2019. 12–21
14 Jiang Y, Zhu Y, Lan C, et al. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, 2020. 463–479
15 Hashemi S H, Jyothi S A, Campbell R H. TicTac: accelerating distributed deep learning with communication scheduling. 2018. ArXiv:1803.03288
16 Jayarajan A, Wei J, Gibson G, et al. Priority-based parameter propagation for distributed DNN training. 2019. ArXiv: 1905.03960

17 Peng Y, Zhu Y, Chen Y, et al. A generic communication scheduler for distributed DNN training acceleration. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019. 16–29
18 Wang G, Venkataraman S, Phanishayee A, et al. Blink: fast and generic collectives for distributed ML. 2019. ArXiv:1910.04940
19 Hochreiter S, Schmidhuber J. Long short-term memory. Neural Comput, 1997, 9: 1735–1780
20 Braun S. LSTM benchmarks for deep learning frameworks. 2018. ArXiv:1806.01818
21 Krueger D, Maharaj T, Kramár J, et al. Zoneout: regularizing RNNs by randomly preserving hidden activations. 2016. ArXiv:1606.01305
22 Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks. In: Proceedings of Advances in Neural Information Processing Systems, 2012. 1097–1105
23 Szegedy C, Liu W, Jia Y, et al. Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015. 1–9
24 He K, Zhang X, Ren S, et al. Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016. 770–778
25 Rumelhart D E, Hinton G E, Williams R J. Learning representations by back-propagating errors. Nature, 1986, 323: 533–536
26 Yu X, Loh N K, Miller W. A new acceleration technique for the backpropagation algorithm. In: Proceedings of IEEE International Conference on Neural Networks, 1993. 1157–1161
27 Kingma D P, Ba J. ADAM: a method for stochastic optimization. 2014. ArXiv:1412.6980
28 Chen T, Li M, Li Y, et al. MXNet: a flexible and efficient machine learning library for heterogeneous distributed systems. 2015. ArXiv:1512.01274
29 Agrawal A, Modi A N, Passos A, et al. TensorFlow eager: a multi-stage, Python-embedded DSL for machine learning. In: Proceedings of the 2nd Conference on Systems and Machine Learning (SysML'19), 2019
30 Collobert R, Kavukcuoglu K, Farabet C. Torch7: a Matlab-like environment for machine learning. In: Proceedings of BigLearn, NIPS workshop, 2011
31 Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. 2014. ArXiv:1409.1556
32 Szegedy C, Vanhoucke V, Ioffe S, et al. Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016. 2818–2826
33 Amodei D, Ananthanarayanan S, Anubhai R, et al. DeepSpeech2: end-to-end speech recognition in English and Mandarin. In: Proceedings of International Conference on Machine Learning, 2016. 173–182
34 Shen J, Pang R, Weiss R J, et al. Natural TTS synthesis by conditioning WaveNet on Mel spectrogram predictions. In: Proceedings of 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2018. 4779–4783
35 Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need. In: Proceedings of Advances in Neural Information Processing Systems, 2017. 5998–6008
36 Wu Y, Schuster M, Chen Z, et al. Google's neural machine translation system: bridging the gap between human and machine translation. 2016. ArXiv:1609.08144
37 Devlin J, Chang M W, Lee K, et al. BERT: pre-training of deep bidirectional transformers for language understanding. 2018. ArXiv:1810.04805
38 Deng J, Dong W, Socher R, et al. ImageNet: a large-scale hierarchical image database. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, 2009. 248–255
39 Panayotov V, Chen G, Povey D, et al. Librispeech: an ASR corpus based on public domain audio books. In: Proceedings of 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2015. 5206–5210
40 Bahrampour S, Ramakrishnan N, Schott L, et al. Comparative study of deep learning software frameworks. 2015. ArXiv:1511.06435
41 Lavin A, Gray S. Fast algorithms for convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016. 4013–4021
42 Xiao W, Han Z, Zhao H, et al. Scheduling CPU for GPU-based deep learning jobs. In: Proceedings of the ACM Symposium on Cloud Computing, 2018. 503–503
43 Chen T, Moreau T, Jiang Z, et al. TVM: an automated end-to-end optimizing compiler for deep learning. In: Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018. 578–594
44 Rotem N, Fix J, Abdulrasool S, et al. Glow: graph lowering compiler techniques for neural networks. 2018. ArXiv:1805.00907
45 Ma L, Xie Z, Yang Z, et al. Rammer: enabling holistic deep learning compiler optimizations with rTasks. In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, 2020. 881–897
46 Jia Z, Padon O, Thomas J, et al. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019. 47–62
47 Chen T, Zheng L, Yan E, et al. Learning to optimize tensor programs. Adv Neural Inf Process Syst, 2018, 31: 3389–3400
48 Zheng S, Liang Y, Wang S, et al. Flextensor: an automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In: Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, 2020. 859–873
49 Vasilache N, Zinenko O, Theodoridis T, et al. Tensor comprehensions: framework-agnostic high-performance machine learning abstractions. 2018. ArXiv:1802.04730
50 Ragan-Kelley J, Barnes C, Adams A, et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. ACM SIGPLAN Not, 2013, 48: 519–530
51 Mirhoseini A, Pham H, Le Q V, et al. Device placement optimization with reinforcement learning. In: Proceedings of the 34th International Conference on Machine Learning, 2017. 70: 2430–2439
52 Sergeev A, Balso M D. Horovod: fast and easy distributed deep learning in TensorFlow. 2018. ArXiv:1802.05799
53 Wu Y, Cao W, Sahin S, et al. Experimental characterizations and analysis of deep learning frameworks. In: Proceedings of IEEE International Conference on Big Data (Big Data), 2018. 372–377
54 Shi S, Wang Q, Chu X. Performance modeling and evaluation of distributed deep learning frameworks on GPUs. In: Proceedings of 2018 IEEE 16th International Conference on Dependable, Autonomic and Secure Computing, 16th International Conference on Pervasive Intelligence and Computing, 4th International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2018. 949–957
55 Wang Y E, Wei G Y, Brooks D. A systematic methodology for analysis of deep learning hardware and software platforms. In: Proceedings of the 3rd Conference on Systems and Machine Learning (SysML'20), 2020
56 Xu P, Shi S, Chu X. Performance evaluation of deep learning tools in docker containers. In: Proceedings of the 3rd International Conference on Big Data Computing and Communications (BIGCOM), 2017. 395–403
57 Shams S, Platania R, Lee K, et al. Evaluation of deep learning frameworks over different HPC architectures. In: Proceedings of IEEE 37th International Conference on Distributed Computing Systems (ICDCS), 2017. 1389–1396
58 Awan A A, Subramoni H, Panda D K. An in-depth performance characterization of CPU-and GPU-based DNN training on modern architectures. In: Proceedings of the Machine Learning on HPC Environments, 2017. 1–8
59 Velasco-Montero D, Fernandez-Berni J, Carmona-Galan R, et al. Optimum selection of DNN model and framework for edge inference. IEEE Access, 2018, 6: 51680–51692