DOI: 10.14529/jsfi190404

Survey on Software Tools that Implement Deep Learning Algorithms on Intel/x86 and IBM/Power8/Power9 Platforms

***Denis Shaikhislamov*[1], *Andrey Sozykin*[2,3], *Vadim Voevodin*[1]**

© The Authors 2019. This paper is published with open access at SuperFri.org

Neural networks are becoming more and more popular in scientific field and in the industry. It is mostly because new solutions using neural networks show state-of-the-art results in the domains previously occupied by traditional methods, eg. computer vision, speech recognition etc. But to get these results neural networks become progressively more complex, thus needing a lot more training. The training of neural networks today can take weeks. This problems can be solved by parallelization of the neural networks training and using modern clusters and supercomputers, which can significantly reduce the learning time. Today, a faster training for data scientist is essential, because it allows to get the results faster to make the next decision.

In this paper we provide an overview of distributed learning provided by the popular modern deep learning frameworks, both in terms of provided functionality and performance. We consider multiple hardware choices: training on multiple GPUs and multiple computing nodes.

Keywords: HPC, neural networks, deep learning frameworks, distributed training.

Introduction

Neural networks are currently one of the most popular methods for creating AI systems. They show state-of-the-art results in many areas, including computer vision [55, 59, 62, 69], natural language processing [53, 58], speech recognition [56, 71]. However, to achieve such results, neural networks are becoming more and more complex and more and more data is needed for their training [53, 60, 64]. As a result, the training of such neural networks requires considerable time: days, weeks, and even months.

Problems with training neural networks can be solved using modern clusters and supercomputers. In this case, the neural network is trained in parallel on several computing nodes of the cluster, which can significantly reduce the learning time [54]. In addition, you can train a large network on a parallel computing system that does not fit in the memory of one computer [52, 60]. As a result, distributed training of neural networks is rapidly gaining popularity.

In this paper, we provide an overview of distributed training of neural networks that exist in modern deep learning frameworks. The usage of various hardware options for distributed training are considered: distributed training on several GPUs and several computing nodes. It also describes approaches to the logic of organizing distributed learning. The most popular approaches of distributed training are model and data parallelism. In data parallelism, each device contains a complete copy of the neural network and performs training on parts of the data. In the case of model parallelism, the neural network is divided into separate parts, which are distributed among devices and are trained on a complete data set.

There are two types of distributed learning organization: asynchronous and synchronous. In asynchronous training, parallelization occurs due to the breakdown of work between workers and parameter servers. Workers are used for training (independently of each other), and parameter servers only store model parameters and their modification. The synchronous approach is

[1]Research Computing Center of Lomonosov Moscow State University, Moscow, Russian Federation
[2]Ural Federal University, Ekaterinburg, Russian Federation
[3]N.N. Krasovskii Institute of Mathematics and Mechanics, Ekaterinburg, Russian Federation

organized as follows. Workers have their own copy of the model, but the data is different. After the workers have processed their part of the data, they exchange results with each other and at the same time change the parameters of the model. You can combine these two methods: use synchronous learning within a node with several GPUs and use asynchronous learning between nodes.

Currently there is a large number of frameworks for training neural networks, the most popular of which are TensorFlow, Caffe, Caffe2, Torch, PyTorch, MXNet, Theano, PaddlePaddle, Microsoft Cognitive Toolkit, Deeplearning4j, Keras and OpenCV. All these frameworks can use parallel training on several computing cores and processors, conduct training on the GPU, but the possibilities of using distributed training are much more limited. In this review, we included libraries that can only parallelize the training of a neural network on several GPUs and computing nodes. Therefore, it was necessary to exclude such popular frameworks as Theano and OpenCV, in which distributed learning tools are not developed.

This paper discusses the possibility of using parallel computing systems only for training deep neural networks. The inference requires other optimizations, including the features of parallel execution and use of GPUs on mobile systems [70]. Supercomputers are rarely used for inference.

The rest of the paper is organized as follows. Section 1 describes the most common frameworks. The following items are highlighted for each framework:

- A brief description of the development history and current status.
- Implementation features and a list of supported algorithms – we give description of the basic principles of the framework. The support for the implementation of the main types of neural networks (fully connected, convolutional (CNN), recurrent (RNN), deep autoencoders (DAE), generative adversarial network (GAN) and networks with Transformer architecture) is indicated for each framework.
- Optimizations for Intel, Power, Nvidia platforms – describes the availability of optimizations, as well as their application in terms of availability.
- Use of several GPUs for training – the possibility of using several GPUs for training both with the standard package and with the help of add-ons.
- Support for training on multiple nodes – the ability to start training a neural network on multiple machines (for example, on different nodes of a cluster). Both standard support and the use of add-ons are considered.

Section 2 shows the comparison of frameworks both in terms of functionality and performance on the base models that are used by the community to evaluate the performance of various frameworks. It is worth noting that in Section 1 the questions of the frameworks performance are not addressed.

1. Deep Learning Frameworks Overview

1.1. TensorFlow

1.1.1. Overview

TensorFlow (hereinafter, TF) [47] is a relatively young framework for high-performance computing, which is mainly used as a framework for implementing deep learning methods and is developed by the Google Brains team. It was developed in a closed manner until 2015, but

after its revamp it was released to the public. The TF core is mostly written in a combination of C++ and CUDA, but there are also parts written in Python. TF has an API for both Python and C++, but since TF is mainly used in Python, this review will look at the Python API.

TF is in an active development. Current version is TensorFlow 2.0, which focuses on ease of use, including the ease of distributed learning. Detailed development plans can be seen in [45].

1.1.2. Implementation features and a list of supported algorithms

TF relies on the concept of a computation graph that describes data and operations on it. Because of this mechanism, TF can be used for almost any calculation. Examples of simple graphs are shown in Fig. 1.

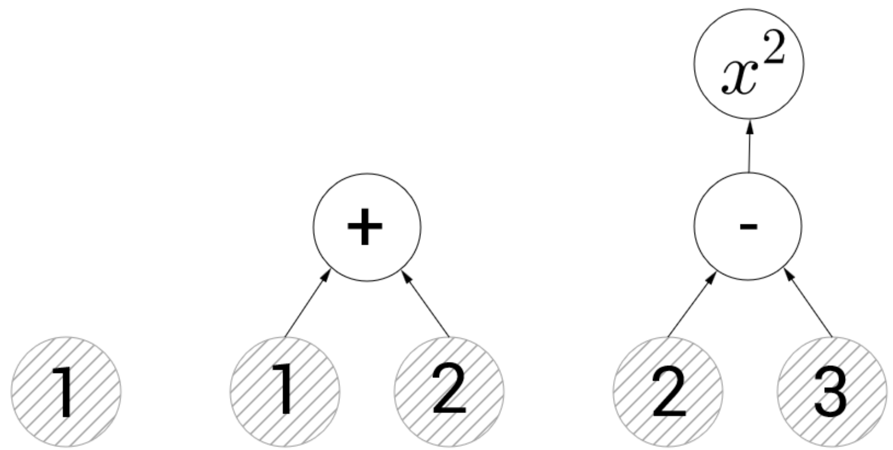


Figure 1. Examples of simple computation grahs

The computation graph can be created by the user himsel. Also, the graph is created by default when the framework is used, which will be used if you do not specify which graph the operation is performed with. TF implements a lot of algorithms for both deep learning and traditional machine learning.

Inside the graph, the data is represented by the so-called tensors (n-dimensional matrices). The size of the matrix can be specified on its definition, or it may not be specified if the size of the matrix changes dynamically during the execution of the calculation graph.

The graph is calculated within the session, to where all the data necessary for calculation is passed, after which the graph is calculated, and the result of the execution is returned. In version TF 2.0, Eager execution has appeared, which makes it possible to start calculations without creating a session.

In TensorFlow you can implement algorithms that efficiently utilize tensors (for example, there are ray tracing implementations on TF). But since all the necessary algorithms for deep learning are provided inside the framework, it is most often used for deep learning. In TF most of the optimizers (gradient descent, Adagrad, AdaDelta, momentum, Adam, etc.) and activation functions (relu, relu6, elu, sigmoid, tanh, softplus, softsign, dropout, softmax etc.) are implemented.

In TensorFlow you can implement all the main types of neural networks: fully connected, convolutional, recurrent, deep autoencoders, GAN and Transformer.

1.1.3. Intel, Power, Nvidia platform optimizations

TF optimization for Intel are mentioned even on the official TF website. For example, TF will work optimally if it is built from the source files provided on the official TF website, which will incorporate all the features of Intel processors. Also, Intel has MKL-DNN, which is supported by TF (TF with Intel MKL-DNN).

TF is available in the PowerAI framework package, which includes Power-optimized versions of the popular frameworks and Distributed Deep Learning (DDL), which is a communication library optimized for distributed training of neural networks. It is worth noting that PowerAI supports only GPU systems.

It is also worth noting that the versions of Intel and PowerAI do not differ in terms of implemented algorithms. This is also true for subsequent frameworks, unless otherwise is specified.

TF has an optimized version for Nvidia GPUs, which can be downloaded from the official website. This version uses the CUDA library, which is used to run general-purpose computing on the GPU, and cuDNN, containing effective operations for working with neural networks on the GPU. In addition, NVidia has a special platform called the Nvidia GPU Cloud, which has a container docker directory with preinstalled and optimized NVidia software, including TensorFlow.

1.1.4. Multi-GPU and Multi-node training of neural networks

The TF framework was developed with the goal of providing distributed training in large clusters, so it includes training tools on several GPUs and computing nodes.

Data-parallel training in TF is implemented in the Distribution Strategy API (tf.distribute.Strategy), which includes several strategies for distributed training [14]. The most popular approach is synchronized distributed training on several GPUs, both on one node (MirroredStrategy) and on several nodes, each of which can have several GPUs installed (MultiWorkerMirroredStrategy). It is also possible to use asynchronous training using the ParameterServerStrategy strategy. The strategy allows the use of several nodes as parameter servers and workers, and nodes with workers can contain several GPUs. When using the Distribution Strategy in conjunction with the high-level TensorFlow APIs (Keras and Estimators), parallelization of training is performed automatically, both on several GPUs within the same node, and between cluster nodes. Developing your own layers or training methods requires you to explicitly use the parallel programming.

Model-parallel training in TF is possible using the Mesh TensorFlow library [66]. This library is an add-on for TF and defines the language of distributed processing of tensors. The library allows you to explicitly indicate by which dimension the tensors will be divided for distributed processing. As a result, it is possible to parallelize operations both by model and by data, which provides data-parallel or model-parallel training, as well as its combination. Training is performed in synchronous mode; all synchronization operations are provided automatically by the Mesh TensorFlow library.

There is a separate library from TF, Horovod [65], which runs on top of TF and PyTorch and provides a simpler interface for implementing distributed synchronous training on clusters. The library uses MPI for communication between processes, and it is also possible to use several GPUs on a node. More details about the use of Horovod can be found in [20]. It is worth noting that Horovod can be used with the DDL communication library from IBM; details are presented in [13].

1.2. Caffe and Caffe2

1.2.1. Overview

Caffe (Convolutional Architecture for Fast Feature Embedding) [57] is a deep learning framework developed by the University of California, Berkeley. The main emphasis in this framework was made on the task of image classification and segmentation. Caffe is distributed under the BSD license. It is believed that Caffe is one of the very first successful deep learning frameworks, considering many research papers that have used this framework. Caffe has a library of pre-built and pre-trained models of neural networks that have been successfully used in a particular subject area. This resource is called Model Zoo and it is considered a big advantage of Caffe, since there are many useful models there, which is why other frameworks usually implement Caffe model converters into their own.

The framework is written in C++ with an interface for Python.

Caffe is no longer supported, since the development team has switched its interest towards the development of Caffe2 [5]. At the end of March 2018, the PyTorch and Caffe2 teams merged, after which Caffe2 moved to the PyTorch repository and became a part of it. The latest version of Caffe, 1.0, was released on April 18, 2017.

1.2.2. Implementation features and a list of supported algorithms

In Caffe, everything is built on layers. Layer is a description of a data processing operation. It can implement both a neural network layer and pooling (non-linear compression of a feature map, in which a group of pixels is compressed to 1 pixel using a non-linear transformation, such as a maximum function), filters, non-linear transformations, an error function, etc. The neural network (Net) itself will consist of many layers, which must be described in the configuration file in the Protobuf format.

The layers also have two functions: forward and backward. In the forward function, the error function is calculated; in the backward function the gradients are calculated to further change the model parameters. Inside the layer, two versions of each of these functions are implemented, one for the CPU, the second for the GPU. Depending on the mode (CPU or GPU), which is specified at the beginning of the program, the corresponding method will be called (CPU is used by default). In the neural network itself, the functions of the forward and backward passage will invoke the corresponding functions of the layers included in the neural network.

The final element is the optimizer (Solver), which implements the optimizer of your choice. Supported: SGD, AdaDelta, AdaGrad, Adam, Nesterovs Accelerated Gradient and RMSProp.

Initially, Caffe implemented algorithms only for machine vision (based on convolutional neural networks), but later added support for recurrent neural networks in the form of LSTM. Now it is possible to implement fully connected, recurrent, convolutional networks, deep autoencoders and GAN. Transformer network implementation was not found.

1.2.3. Intel, Power, Nvidia platform optimizations

Intel provides Intel Distribution of Caffe – a version optimized for Intel processors [24]. This version of Caffe uses Intel MKL-DNN, which, in its turn, allows you to use OpenMP by default.

Caffe2 worked together with Intel to integrate Intel MKL functions to optimize the performance of the framework when used in production, but at the moment Caffe2 does not have full

MKL support. MKL still can be used with Caffe2, but some users had problems with neural network training using Caffe2 with Intel MKL.

Caffe is included in PowerAI. There are two versions of Caffe in PowerAI: Caffe BVLC – a standard version of Caffe developed in BVLC (Berkeley Vision and Learning Center); Caffe IBM is an IBM-optimized version of the framework. The default version is Caffe IBM. Caffe2 is included in PyTorch, which in turn is included in PowerAI.

Caffe and Caffe2 support CUDA and CuDNN, and also have containers in the Nvidia Cloud.

1.2.4. Multi-GPU training of neural networks

To use several GPUs for training in Caffe, you must use your own Caffe scripts; a complete guide is provided in [4]. Synchronized distributed training is used. Asynchronous distributed training, as well as model parallelism, are not supported.

You can use a special version of Caffe, NVCaffe, which is supported by NVidia. This version was created specifically for the use of several GPUs. User instructions can be found in [35].

Caffe2 has a special distributed training module that implements various algorithms (for example, SynchronousSGD) of synchronous distributed training models. Caffe2 uses NCCL for synchronization, a communication library for multi-GPU nodes, which provides very good scalability. An example of use is given on the official website for training the ResNet-50 neural network both at several GPUs and at several nodes in [6]. A more general instruction is provided in [7]. Model parallelism in Caffe2 is possible, but manual distribution of neural network layers across devices is required before training the model.

Caffe2 has no official support for asynchronous training.

1.2.5. Multi-node training of neural networks

When using Intel Distribution of Caffe, [19] provides guide on setting up and starting the training process. Multi-node mode uses synchronized distributed training. For synchronization after processing their data, the nodes use the Intel Machine Learning Scaling Library (MLSL) for communication, which implements communication primitives for both model and data parallelism. But for multi-node training you need to configure the nodes separately, therefore the problem: the user may not know which nodes his application will work on.

For HPC clusters with GPU nodes, Inspur developers developed a version of Caffe – Caffe MPI [3] – which utilizes MPI to synchronize results between nodes after processing its part of the data.

In Caffe2, similar to multi-GPU training, you can also run model training on multiple nodes. You can select several communication backends to synchronize the results: Gloo and Redis. It is also worth noting that it is necessary for all nodes to have a common shared folder through which, after starting the program, the nodes can detect each other and start the training. More detailed instructions for using this mode are given in the ResNet-50 training example in [6].

1.3. Torch

1.3.1. Overview

Torch [51] – MATLAB-like framework for the Lua programming language, which provides a huge set of algorithms for deep learning. The core of the framework is written in C, but the

programs themselves are written in Lua. LuaJIT is used for JIT compilation, which significantly speeds up the program. Framework is distributed under the BSD license. Despite the fact that Torch is a very powerful and convenient framework, programs for the framework must be written in Lua. As noted in [31], Lua is not a popular programming language in the field of data science. Users work around this problem by using the PyTorch package, which has an interface for Python and not only includes all the functionality of Torch, but also complements it.

Torch is not in active development, as the team switched to PyTorch development. The latest version, torch7, was released on February 27, 2017.

1.3.2. Implementation features and a list of supported algorithms

The core of the framework is in the torch package, in which there is a tensor implementation – similar to TensorFlow, n-dimensional arrays, basic indexing operations, getting array slices, transposing, etc., as well as BLAS operations, implementation of mathematical functions (max, min, etc.) and statistical distributions.

The next important package is nn, which is used to build neural networks. The package has many different parts, but every one of them has a common part called "Module", which implements the forward and backward functions, that allow you to make forward and backward passes through the network. Modules can be connected using classes such as Sequential, Parallel, Concat, which allows you to build complex models of neural networks. There is also a list of basic modules, such as Tanh, Linear, Max, etc.

Error functions are implemented as subclasses of the Criterion class, which is similar to the Module class, since it has the same forward and backward functions. Torch include implementations of the basic error functions, such as cross-entropy, mean squared error, and stochastic gradient descent.

Torch allows you to implement all the most popular types of neural networks: fully connected, convolutional, recurrent, deep autoencoders, GAN and Transformer.

1.3.3. Intel, Power, Nvidia platform optimizations

Torch has a separate development branch that has Intel optimizations, especially for Intel Xeon. You can use Intel MKL in conjunction with Torch. The repository branch is located in [25].

Torch is included in the PowerAI framework.

Optimizations for the NVIDIA platform are included in cutorch – the CUDA backend for Torch. Torch supports CUDA and cuDNN. Nvidia Cloud does not contain a container with an optimized version of Torch.

1.3.4. Multi-GPU training of neural networks

Support for multiple GPUs is included in the standard version of Torch. To utilize multi-GPU training, you need to use the module in nn – DataParallelTable – which allows you to use data parallelism. Torch supports synchronous distributed training. It should be noted that parallelization is done quite simply: you just need to wrap the Module that implements the neural network in a DataParallelTable with the list of GPUs on which the training will be conducted. A more detailed guide can be found in [9].

There is no official support for asynchronous training and model parallelism with Torch.

1.3.5. Multi-node training of neural networks

Torch does not have official support for distributed training on multiple nodes, but there is a separate development branch, Torch MPI, which allows you to use nodes not only exclusively with the CPU, but also with the GPU. More detailed information can be found in [46].

1.4. PyTorch

1.4.1. Overview

PyTorch [42] is a deep learning framework that was based on Torch and is developed primarily by the Facebook's artificial intelligence research group. PyTorch also includes the Caffe2 framework, so PyTorch has advanced distributed training capabilities.

Pytorch is currently under active development. The stable version, 1.3.0, was released on October 10, 2019.

1.4.2. Implementation features and a list of supported algorithms

Due to the fact that PyTorch is based on Torch, they are very similar. The main module, torch, includes tensors and operations (transformations, mathematical functions, etc.), serialization operations.

The next important module that is used to build neural networks: torch.nn. This module includes a description of the base class Module, which is inherited by all types of neural network layers implemented in PyTorch (convolutional, pooling, linear, recurrent, etc.). The module has implementations of a large number of activation functions (for example, ReLU6, sigmoid, softsign, tanh, etc.), error functions (MSE, L1Loss, CrossEntropy, BCELoss, etc.). Each module has, similar to Torch, forward and backward functions. The neural network itself is a Module object, that is a combination of other Module, which allows to automatically make forward and backward passes through the resulting network.

With PyTorch, you can implement almost all the types of architecture used in neural networks: fully connected, convolutional, recurrent, deep autoencoders, GAN, and Transformer.

1.4.3. Intel, Power, Nvidia platform optimizations

Intel optimizations mainly consists of integrating Intel MKL-DNN into PyTorch [22]. Prior to version 1.0, the PyTorch distribution did not utilize Intel MKL-DNN, but now PyTorch supports it by default.

IBM has included PyTorch in PowerAI, which uses OpenBLAS.

For NVidia, PyTorch is supported on Nvidia Cloud. It is also worth noting that there was a collaboration between PyTorch and Nvidia, which resulted in Apex – A PyTorch Extension – a set of utilities that make it easy to use distributed training technologies and Automatic Mixed Precision. PyTorch supports CUDA and CuDNN.

1.4.4. Multi-GPU training of neural networks

Just like Torch, PyTorch has a DataParallel module in the nn package, and if you wrap a neural network model in this module, then the training will be done on several GPUs. It uses data parallelism and synchronous distributed training. Model parallelism is also possible: when

defining a model, it is necessary to distribute the layers across different devices and implement its forward and backward pass function, including data transfers between GPUs [43].

PyTorch has no official support for asynchronous training.

1.4.5. Multi-node training of neural networks

PyTorch has a DistributedDataParallel module that uses the torch.distributed package to parallelize training across multiple processes/nodes. This module can be used in two modes: 1 process with several GPUs and several processes with 1 GPU. Developers do not recommend using the use case with several GPUs, noting that the framework will work faster if you create 1 process for each GPU. The module supports 2 backends for communication: gloo and nccl. Developers recommend using the nccl backend, as it showed better performance in their experiments.

In version 1.0, developers changed the core of the torch.distributed module, and now it depends on the C10D library, which works asynchronously for all supported backends. It sped up all of its dependent modules (DataParallel, DistributedDataParallel).

It is also worth noting that Horovod supports PyTorch. Instructions can be found in [20].

1.5. Apache MXNet

1.5.1. Overview

MXNet [1] is a popular deep learning framework known for its flexibility and ability to scale across multiple GPUs and nodes. It is developed by a team from the Apache Software Foundation and distributed under the Apache 2.0 license. The core is written mainly in C++; there are interfaces for a large number of programming languages: C++, Python, R, MatLab, JavaScript, Go, Scala, Perl, Julia, Wolfram Language. Due to its scalability, Amazon has chosen MXNet for its AWS cloud environment. Like Caffe, MXNet has its own library of pre-trained models - Gluon model zoo.

MXNet is currently under active development. The latest version, 1.5.0, was released on June 8, 2019.

1.5.2. Implementation features and a list of supported algorithms

MXNet is similar to TensorFlow: it operates on NDArray (similar to a tensor, it is an n-dimensional array) and a computation graph. Graphs include variables - objects whose type and size are not determined during its initialization, but will be calculated as the data is fed into the graph.

A neural network is built using the Module API, which implements almost all common types of layers of neural networks, activation functions and optimizers.

MXNet has a higher level and simpler interface for creating neural networks - Gluon. Gluon is very similar to Keras API in its simplicity of neural network creation. An initial guide to using Gluon can be found in [2].

MXNet supports most popular neural network architectures such as fully connected, convolutional, recurrent, deep autoencoder, GAN, and Transformer (using the GluonNLP extension).

1.5.3. Intel, Power, Nvidia platform optimizations

Intel and Apache MXNet released version 1.2.0, in which the main point was to optimize MXNet for CPUs using Intel MKL-DNN, which significantly accelerated the work of the framework. Detailed information about both the installation and how MXNet has accelerated can be found in [17].

Authors could not find the information about optimizing the framework for the Power architecture. MXNet is not part of PowerAI but installing MXNet is still possible.

For NVidia, MXNet is supported by Nvidia Cloud. MXNet also has support for CUDA and CuDNN.

1.5.4. Multi-GPU training of neural networks

By default, data parallelism is used in MXNet, but model parallelism is also supported. [34] provides an example of using several GPUs for parallelizing a LSTM model.

Data parallelism is quite simple. When initializing the module, it is necessary to give a list of GPUs on which training will be conducted. There is also built-in support for static load balancing: if one GPU is faster than another, then you can set the proportions in which the GPUs will process the data. Detailed information on GPU parallelization can be found in [32, 33].

1.5.5. Multi-node training of neural networks

MXNet officially supports both asynchronous and synchronous distributed training. For distributed training, MXNet uses three kinds of processes. The first of these is a worker in which training will occur, and which can use multi-GPU training. The second process is called the server (similar to the parameter server in Tensorflow), which stores the model parameters. There can be several servers, and they can be located both on the machine with the worker or on another machine. Servers store parameters in a key-value format. The third type of process is the scheduler, which initializes the cluster and is responsible for ensuring that other processes can interact with each other.

The training depends on which mode the server is created with. There are 4 modes: dist_sync – for synchronous training; dist_async – for asynchronous training; dist_sync_device – similar to dist_sync, but is used for training on several GPUs, which allows to skip time-consuming communications between the CPU and GPU and synchronize the results only between GPUs (this method uses more memory on the GPU); dist_async_device – similar to dist_async, but is used for training on several GPUs.

If the communication becomes a bottleneck, you can use compression of the gradients to reduce the load on the communication.

It is also worth noting that MXNet added integration with Horovod for distributed training in 1.4.0 version.

1.6. PaddlePaddle

1.6.1. Overview

PaddlePaddle [36] is a deep learning framework introduced by Chinese search giant Baidu in 2016. The main highlighted features are simplicity, scalability and flexibility. It is written in C++, but it also has interfaces for C++, Python.

PaddlePaddle is currently under development. The latest version, PaddlePaddle 1.5.1, was released on July 16, 2019.

1.6.2. Implementation features and a list of supported algorithms

The main concept in PaddlePaddle, like TensorFlow, is a computation graph, but the operation is different: in PaddlePaddle a Python program that describes a neural network model builds a computation graph in protobuf format [41], and then sends it for execution to the so-called Executor: either process responsible for distributed training, or to the libpaddle.so library for local execution. A description of how the architecture works and what were the reasons for such a design can be found in [45].

PaddlePaddle supports most of the neural network architectures used: convolutional, recurrent, fully connected, deep autoencoders, GAN, and Transformer.

1.6.3. Intel, Power, Nvidia platform optimizations

PaddlePaddle, like other frameworks, can use Intel MKL to speed up the framework on the CPU and Intel MKL-DNN to speed up the convolutional neural networks.

Information about optimizing the library for the Power IBM architecture could not be found.

PaddlePaddle is supported on Nvidia Cloud and can also utilize CUDA and CuDNN.

1.6.4. Multi-GPU training of neural networks

PaddlePaddle has official support for multi-GPU training and has two ways to use it. The first way is that ParallelExecutor is used instead of the usual Executor. PaddlePaddle uses synchronous distributed training. Information on the implementation of asynchronous training in official sources could not be found. More detailed instructions can be found in [39]. Developers state that current version of Fluid provides only data parallelism training mode [38].

The second way is to compile the computation graph using CompiledProgram, which transforms the graph for faster execution. Then you need to call with_data_parallel, which transforms the graph so that several devices (in CPU mode, threads) can be used. PaddlePaddle automatically detects all available GPU devices and distributes the work between them. Developers recommend using this method. An example of usage can be found in [40].

1.6.5. Multi-node training of neural networks

PaddlePaddle has official multi-node support, and there are two possible uses. In case of the RPC communication backend, training is based on the Trainer-ParameterServer architecture. Both synchronous and asynchronous distributed training are supported. Trainer is engaged in the training, and the ParameterServer is responsible for storing and modifying the model parameters. The peculiarity is that it is necessary to start the processes of parameter servers and the training processes separately, in which it is necessary to specify the address and port of the parameter server. DistributeTranspiler is used to distribute training, which, depending on how many trainers and parameter servers, can give individual processes the computation graph that they need. The output computation graphs also contain all the routines of synchronization and parameter updates.

In case of NCCL communication backend, the parameter server is no longer needed, since the trainers themselves are communicating and updating model parameters. All the examples of multi-node training can be found in [38].

1.7. Microsoft Cognitive Toolkit

1.7.1. Overview

Microsoft Cognitive Toolkit [29] (hereinafter, CNTK) is a deep learning framework developed by Microsoft Research. It is written in C++ and has interfaces in C++, Python and BrainScript.

The developers stopped developing this framework after version 2.7.0, released on March 29, 2019.

1.7.2. Implementation features and a list of supported algorithms

CNTK is similar to the Keras API in building neural networks. CNTK provides basic modules that implement the most used units in neural networks.

The cntk.layers module includes various types of layers of neural networks: recurrent, Embedding, Dense, etc. Basic models, which consist of a sequence of layers of different or identical types, can be built using Sequential. The cntk.learners module provides set of the most popular optimizers that are used in practice, such as SGD, Adam, Nesterov, RMSProp, etc. The cntk.losses module implements error functions like binary cross-entropy, squared error, etc.

After constructing the neural network model from the provided blocks, it is necessary to create a Trainer object, which receives the model at the input and the selected optimizer for training the model. Then you can start training the model using train_minibatch.

CNTK supports such types of neural network architectures as fully connected, convolutional, recurrent, deep autoencoders, GAN and Transformer.

1.7.3. Intel, Power, Nvidia platform optimizations

Microsoft CNTK has two versions: CPU-only, which uses Intel MKL-DNN by default, and a version with a GPU that uses CUDA (CUB and cuDNN).

The only mention authors found about CNTK for Power platform is in Keras docs: Keras supports the CNTK backend.

CNTK is supported in the Nvidia Cloud.

1.7.4. Multi-GPU training of neural networks

CNTK has distributed training support. Synchronous (DataParallelSGD, BlockMomentumSGD, ModelAveragingSGD) and asynchronous (DataParallelASGD) optimizers can be used. The prerequisite for distributed training is that you need to install the MPI for communication between processes. CNTK does not support other communication backends (for example, you can use Gloo with Caffe2). It is worth noting that multi-GPU training on a node occurs through MPI, where each process uses 1 GPU, and processes are communicating through MPI.

Parallelization is quite simple – after the definition of the selected optimizer, you need to wrap it in data_parallel_distributed_learner. A detailed guide to using the distributed CNTK package can be found in [30].

CNTK does not support model parallelism.

1.7.5. Multi-node training of neural networks

Multi-node training is completely same as the multi-GPU training, since MPI is used for communication between processes for several GPUs as well as for several nodes.

1.8. Deeplearning4J

1.8.1. Overview

Deeplearning4j [10] (hereinafter, DL4J) is a deep learning framework written in Java. It has interfaces for Java, Scala, Python (via Keras), Clojure. It is the first large-scale deep learning library for Java, and provides a convenient and scalable interface for distributed training when used with the default integration with Hadoop and Spark. DL4J needs an additional ND4J library if the GPU computing is needed that implements CUDA. DL4J is distributed under the Apache 2.0 license.

The development of the framework is quite slow. The latest stable version, 0.9.1, was released on August 12, 2017, and there is also a beta version 1.0.0-BETA4.

1.8.2. Implementation features and a list of supported algorithms

There are two ways to work with the framework. The first way most people deal with when they start working with this framework is the usage of MultiLayerNetwork. This is a high-level API for building neural networks consisting of a sequence of layers of a certain type. The syntax for this API is very similar to the Keras API.

The second method is the manual construction of a computation graph that describes the architecture of the neural network. It is worth noting that everything that can be done through MultiLayerNetwork can also be done by constructing a graph of calculations, but the configuration of this graph will be much more complicated. However, this approach allows you to implement any desired network architecture.

For data processing, the DataVec module is included in the framework, which implements almost all the necessary functions for loading, saving and converting data, and developers recommend its use wherever possible.

DL4J uses an additional ND4J library to work with tensors, which provides the ability to work with n-dimensional arrays, as well as the ability to use not only CPUs for processing, but also GPUs.

The framework supports most of the popular types of neural network architectures: convolutional, recurrent, fully connected, deep autoencoders. In DL4J there is no way to create a GAN by your own means, but you can import the GAN model described in Keras. Transformer implementation in DL4J could not be found.

1.8.3. Intel, Power, Nvidia platform optimizations

Like other frameworks, DL4J can utilize Intel MKL BLAS.

DL4J was optimized for Power platforms, the process of which is described in [11], but the information is outdated, since the page to which the link had been provided in the article to the optimized version of the library no longer exists.

NVidia GPUs can be used in DL4J via ND4J library, which supports the CUDA and cuDNN libraries. DL4J is not supported in Nvidia Cloud.

1.8.4. Multi-GPU training of neural networks

DL4J supports multi-GPU training, and the parallelization process is quite simple and transparent for the developer: after defining the model using, for example, MultiLayerNetwork, you need to pass it to the ParallelWrapper and start the training process. This wrapper implements synchronous distributed training; synchronization of model parameters are implemented in the wrapper. A detailed guide can be found in [16]. It is claimed that DL4J supports model parallelism, but the authors could not find any guides.

1.8.5. Multi-node training of neural networks

Multi-node training, similarly to PaddlePaddle and MXNet, takes place according to the parameter server - worker architecture, in which the worker processes their part of the data, sends the results to the parameter server, which, after modifying the parameters, sends all updated model parameters to everyone. All examples of usage provided on the official website are designed to work on Spark clusters [12, 68].

1.9. Keras

1.9.1. Overview

Keras [50] is an open library used to work with neural networks that was written in Python. The library was developed as part of the research efforts of the ONEIROS project and is an add-on to other frameworks (front-end) for deep learning. The basic principle followed by the developers is to make the interface between the developer and the backend as intuitive and convenient as possible for quick development. Keras supports the following frameworks as a backend: TensorFlow, Theano, CNTK, MXNet.

Keras is under active development. The latest version, 2.2.4, was released on October 3, 2018. In addition, Keras has been included with TensorFlow and has been recommended to use as a high-level API since TF 2.0. The TensorFlow version of Keras includes a large number of optimizations for TensorFlow that are not found in the standalone version of Keras.

1.9.2. Implementation features and a list of supported algorithms

There are two ways to build neural network models in Keras - using the Sequential class or functional API.

Sequential is a tool for building neural networks in which layers go sequentially one after another. Keras implements a large number of layers of neural networks, such as LSTM, layers for convolutional neural networks (Conv1D, Conv2D, Conv3D), etc. After building the model, you need to compile it: call the compile method with the specified optimizer and error function. Keras provides an extensive set of implemented optimizers, such as SGD, Adam, Nadam, Adagrad, RMSProp, etc.

Using the functional API makes it possible to implement more complex types of neural networks in which layers can connect to each other arbitrarily, including several layers that work in parallel. One example of using the functional API is to build a neural network with multiple inputs. This neural network receives part of the input data at the input of the first layer, and the next part of the input data is supplied to the neural network only after merging

with the output of one of the internal hidden layers. A more detailed guide to the functional API and examples of its use can be found in [18].

Keras supports most of the popular neural network architectures: convolutional, recursive, fully connected, deep autoencoders, GAN and Transformer.

1.9.3. Intel, Power, Nvidia platform optimizations

Due to the fact that Keras is the interface between the developer and the backend, all the optimizations applied to the framework chosen as the backend are also applicable to Keras.

At Nvidia Cloud Keras is available as part of the container with TensorFlow.

1.9.4. Multi-GPU training of neural networks

Keras officially supports multi-GPU training, but only with the TensorFlow backend using the Distribution Strategy API.

It is also possible to use distributed training with the MXNet backend. To use the MXNet backend, you must use the Keras version supported by MXNet developers [26]. When compiling the model, it is necessary to pass context parameter to the input of the method, which contains a list of GPUs on which training can be conducted. A more detailed usage guide is provided in [44].

There is no support for model parallelism in Keras.

1.9.5. Multi-node training of neural networks

In Keras with the TensorFlow backend, multi-node training support is available in an experimental mode with the MultiWorkerMirroredStrategy strategy. Alternatively, you can use the Horovod framework, which has good support for Keras models.

2. Frameworks Comparison

2.1. Basic Comparison

Table 1 provides a comparison of frameworks by common parameters. The designations used in the application: sync / async – support for synchronous / asynchronous distributed training, "?" in the column about the availability of optimizations for the platform – the authors could not find information on the highlighted items. A + in the column about the maximum number of GPU / nodes on which training was started means that there is support for the specified mode, but the authors could not find quantitative results.

In terms of the supported types of neural networks, the frameworks are very close. Almost all frameworks, excluding only Caffe and DeepLearning4J, support the popular neural network architectures: fully connected, recurrent, deep autoencoders, GAN, and Transformer. Caffe and DeepLearning4J do not support architectures that have become popular relatively recently: GAN and Transformer.

Table 1. Basic comparison of frameworks

Title	Written in	Available interfaces	Supported types of neural networks					Supported modes of distributed training	Distributed training models	The maximum number of GPU/nodes on which training was run	Optimizations for		
			CNN	RNN	DAE	DAE	Transformer				Intel	Power	NVidia CUDA/ Nvidia GPU Cloud
TensorFlow	C++, CUDA, Python	Python, C/C++, Java, Go, JS, R, Julia, Swift	+	+	+	+	+	Sync/ acync	Data parallel, model parallel	8 / 64[4]	+ (MKL BLAS + DNN)	+	+/+
Caffe	C++	C++, Python	+	+	+	+	-	sync	Data parallel	256/64[5]	+ (MKL BLAS + DNN)	+	+/+
Caffe2	C++	C++, Python	+	+	+	+	-	sync	Data parallel, model parallel	+/+	+ (MKL BLAS)	+	+/+
Torch	C, Lua	Lua, C	+	+	+	+	+	Sync/ async	Data parallel	256/64[6]	+ (MKL BLAS + Xeon optimizations)	+	+/-
PyTorch	Python, C, Lua	Python	+	+	+	+	+	sync	Data parallel, model parallel	8/8	+ (MKL-DNN)	+	+/+
MXNet	C++	C++, JS, Julia, Go, Python, R, Scala, Perl, Matlab	+	+	+	+	+	Sync/ async	Data parallel, model parallel	256/16[7]	+ (MKL-DNN)	-	+/+
PaddlePaddle	C++, Python	Python	+	+	+	+	+	sync	Data parallel	+/+	+ (MKL BLAS + DNN)	-	+/+
Microsoft Cognitive Toolkit	C++	Python, C++, BrainScript	+	+	+	+	+	Sync/ async	Data parallel	8/8	+ (MKL-DNN)	-	+/+
Deeplearning4j	C++, Java	Java, Scala, Clojure, Python, Kotlin	+	+	+	-	-	sync	Data parallel	+/+	+ (MKL BLAS + DNN)	?	+/-
Keras	Python	Python, R	+	+	+	+	+	Sync/[8] async	Data parallel	+/+[9]	+ (MKL BLAS + DNN)	+	+/-

Most of the frameworks are implemented in C++ for high performance but provide APIs in many languages. The most popular API is for Python, which allows you to quickly develop prototypes of neural networks. An exception is the Torch framework, which is written in C and Lua and provides APIs in the same languages. However, Torch is now inferior in popularity to PyTorch, which provides a Python API. Another exception is the DeepLearning4J framework, which is written in Java and provides an API for languages that use the JVM. Thanks to this, DeepLearning4J integrates well in distributed computing systems from the JVM ecosystem: Hadoop and Spark.

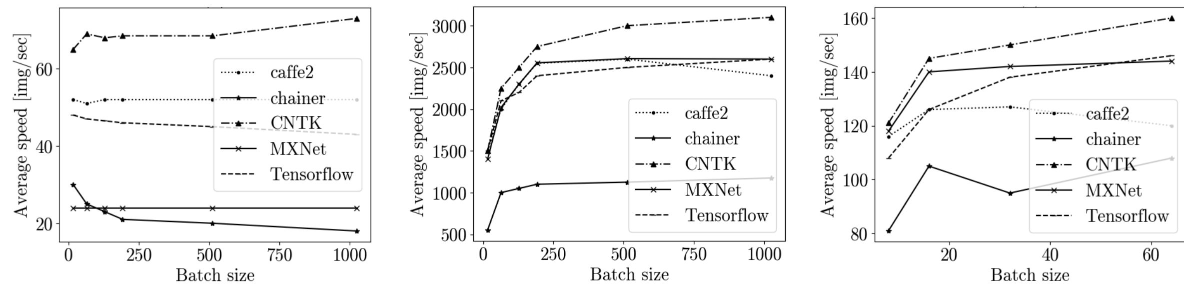
All frameworks support synchronous distributed training, while TensorFlow, Torch, MxNet, CNTK and Keras additionally support asynchronous. Also, all frameworks provide the possibility of distributed training using data parallelism, and TensorFlow, Caffe2, MxNet and Pytorch – using model parallelism. Thus, TensorFlow and MxNet have the advantage in the amount of distributed training modes.

[4]by using Horovod [28]
[5]PowerAI, 4GPU/node
[6]PowerAI, 4GPU/node
[7]16 entities of AWS P2.16x1
[8]Asynchronous training is only possible with pure TF after model conversion
[9]Distributed multi-node training through the use of pure TF or Horovod

2.2. Scalability Comparison

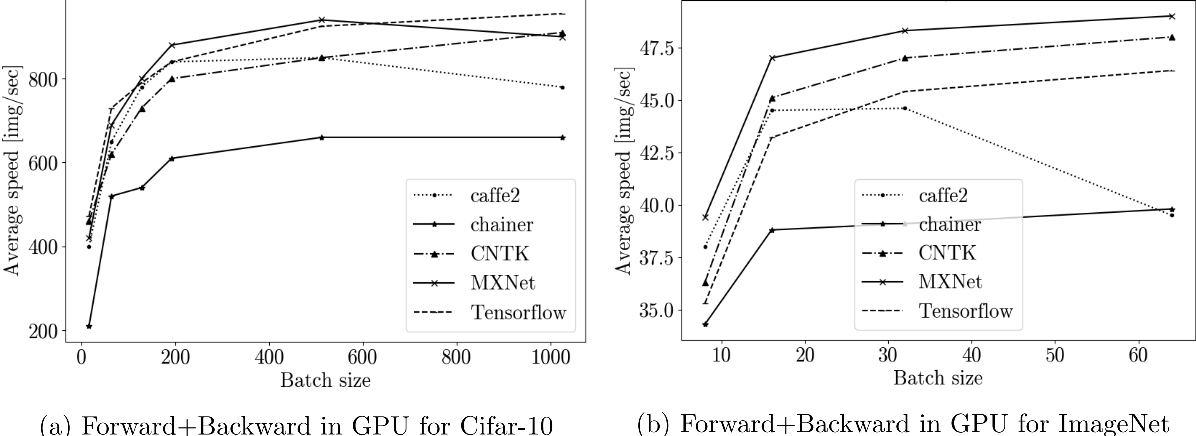
In a review of frameworks for distributed deep learning [63], the authors looked into the simplicity of parallelization, as well as performance on both several GPUs and several nodes. For testing, the authors used the AWS P2.xlarge cloud platform, where each entity was equipped with one NVIDIA Tesla K80 and four 2.7GHz Intel processors. For communication between nodes, 10Gbps Ethernet is used, which can greatly affect their performance, but the authors argue that if all the training data was downloaded to the nodes beforehand, then this communication network is sufficient to transfer model parameters. Authors selected CNTK, TensorFlow, Caffe2, MXNet, and Chainer (not covered in this review). Also, OpenMPI 3.0.0 was used. All experiments were conducted for the ResNet50 neural network; Cifar-10 and ImageNet were used as data for training and testing.

As mentioned earlier, the article looked into the simplicity of the framework for distributed training. The authors concluded that TensorFlow has the most complex architecture and parallelization methods for the user, which is why TensorFlow is excluded in some tests. There is only one scenario with the CPU – for inference, all other scenarios use the GPU.



(a) Forward in CPU for Cifar-10 (b) Forward in GPU for Cifar-10 (c) Forward in GPU for ImageNet

Figure 2. Frameworks performance for image classification (inference, measured in images/sec)



(a) Forward+Backward in GPU for Cifar-10 (b) Forward+Backward in GPU for ImageNet

Figure 3. Frameworks performance on GPUs, measured in images/sec

As you can see in the figures above, Chainer loses everywhere in terms of performance. On the CPU side, CNTK is clearly ahead of the competition in terms of performance, which shows how much this framework is optimized for working on the CPU. On the GPU side, all frameworks except Chainer show good performance, but you can highlight CNTK in the classification, and MXNet in training – these frameworks showed the best results.

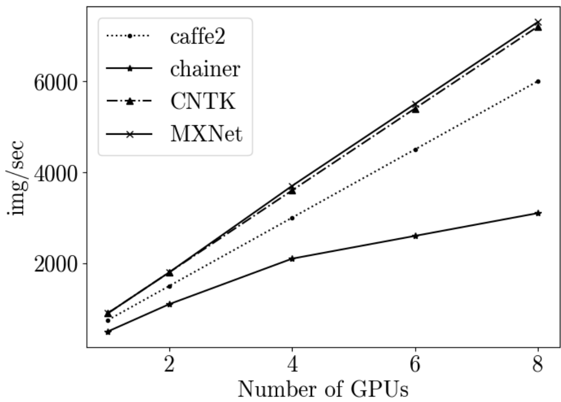
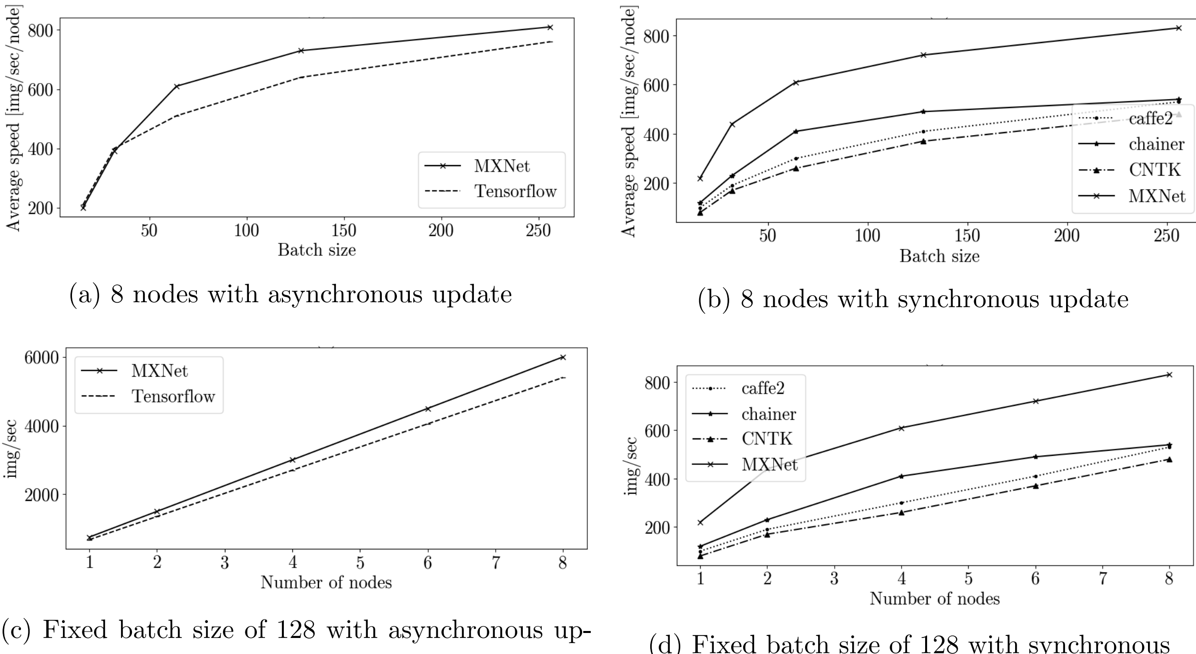


Figure 4. Performance with synchronous distributed multi-GPU training on Cifar-10

Figure 4 shows the performance of distributed multi-GPU training. In the version of TensorFlow, used by the authors, it was necessary to manually implement the mechanism for updating parameters after synchronization, which is why the authors excluded the framework from this test. As you can see, CNTK and MXNet clearly stand out by showing near-linear acceleration and having better performance. The scalability of Caffe2 is also good, but it lags behind due to the training speed of 1 GPU, which is lower than that of competing frameworks.



(a) 8 nodes with asynchronous update

(b) 8 nodes with synchronous update

(c) Fixed batch size of 128 with asynchronous update

(d) Fixed batch size of 128 with synchronous

Figure 5. Performance with distributed multi-node training for ResNet-50 on Cifar-10

Figure 5 shows the results of multi-node training performance. The authors tested scalability to just 8 nodes (1 GPU per node), but it still gives an idea of which framework scales better. TensorFlow was excluded from the tests with synchronous training due to the lack of a ready-made solution for this use case. As you can see from the graphs, MXNet is the clear leader, which is far ahead of all the frameworks in all tests, showing scalability close to linear.

Table 2. The processing speed of the batch on a different number of GPUs for various neural network architectures, in sec

		# of GK210		
		1	2	4
FCN-R	Caffe	0.239	0.131	0.094
	CNTK	0.1811	0.111	0.072
	TF	0.208	0.144	0.121
	MXNet	0.184	0.104	0.86
	Torch	0.165	0.110	0.112
AlexNet-R	Caffe	0.137	0.085	0.047
	CNTK	0.108	0.062	0.037
	TF	0.385	0.332	0.321
	MXNet	0.122	0.070	0.041
	Torch	0.141	0.077	0.046
ResNet-56	Caffe	0.378	0.254	0.177
	CNTK	0.562	0.351	0.170
	TF	0.523	0.291	0.197
	MXNet	0.270	0.167	0.101
	Torch	0.301	0.182	0.096

In terms of scalability, MXNet generally shows the best results. However, the authors also checked the quality of the models in [63], since a higher speed of the framework does not mean that the quality of the resulting model will be higher. The training speed of MXNet is 1.5+ times higher than that of TensorFlow; however, MXNet in 30 epochs cannot achieve the accuracy on the test set, which TensorFlow reaches in 5, which indicates a very fast convergence in TensorFlow. Caffe2 can also be highlighted, which shows good convergence (the next after TensorFlow), but which in terms of training speed is very close to MXNet (faster than TensorFlow by around 1.5 times).

A similar review was made in [67], where several neural network architectures were selected and CNTK, MXNet, Caffe, Torch, and TF were tested. This article did not use multi-node training, only multi-GPU training (maximum 4). The processing speed of one batch on several GK210 GPUs is shown in Tab. 2. For AlexNet-R and ResNet-56, Cifar-10 was used as data, for FCN-R – MNIST. According to the results, you can see that TF scales very poorly on this platform. Torch and CNTK are the best in scalability, but despite this, Torch loses in speed to both CNTK and MXNet.

In terms of the rates of convergence, authors drew a conclusion, that Torch and CNTK successfully cope with FCN-R; MXNet with Torch show the best performance with AlexNet-R and ResNet-56.

In [49] authors explored Power AI DDL, which shows the results of the IBM-Caffe and Torch versions provided in PowerAI on ResNet-50 (ImageNet with 1K classes was used). The results are shown in Tab. 3 and 4.

You can notice that these frameworks from PowerAI show very good scalability on Power IBM platforms, and the frameworks were tested on a large number of nodes, which is very rare.

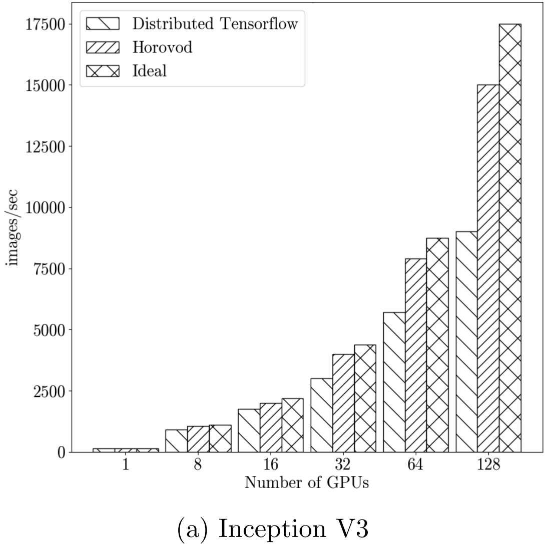
[28] shows the scalability of TF using the Horovod frontend for distributed training on two neural network architectures: Inception V3 and ResNet-101. The results are presented in Fig. 6. Details of what data and GPU are used were not indicated. It is worth noting that Horovod not

Table 3. Caffe multi-node training performance

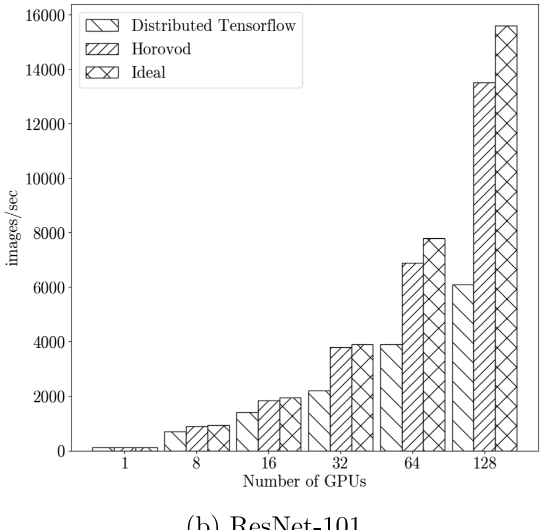
#GPUs	4	8	16	32	64	128	256
#Nodes	1	2	4	8	16	32	64
Speedup	1.0	2.0	3.9	7.9	15.5	30.5	60.6
Scaling efficiency	1.00	1.00	0.98	0.99	0.97	0.95	0.95

Table 4. Torch multi-node training performance

#GPUs	4	16	64	96	128	192	256
#Nodes	1	4	16	24	32	48	64
Speedup	1.0	3.5	13.7	20.5	27.2	40.3	53.7
Scaling efficiency	1.00	0.86	0.86	0.85	0.85	0.84	0.84



(a) Inception V3



(b) ResNet-101

Figure 6. Multi-GPU distributed training performance of Horovod+TF

Table 5. Training speed comparison of PaddlePaddle and other Deep Learning Frameworks on selected models, in img/sec

	SE-ResNeXt50		**YOLOv3**	
	Paddle 1.5.0	PyTorch 1.1.0	Paddle 1.5.0	MXNet
1 GPU	168.334	163.130	29.901	18.578
8 GPUs	843.348	595.274	58.175	35.574
	DeepLab V3+		**Transformer**	
	Paddle 1.5.0	TensorFlow 1.12.0	Paddle 1.5.0	TensorFlow 1.12.0
1 GPU	13.695	6.4	4.865	4.750
8 GPUs	59.721	16.508	4.227	2.302

only provides more impressive results, but also has a more convenient and simple parallelization interface than the default TensorFlow.

Developers of PaddlePaddle benchmarked and compared their framework with PyTorch, MXNet and TensorFlow [37]. They focused on testing single-node multi-gpu Distributed Training and used SE-ResNeXt50, Mask-RCNN, DeepLab V3+ etc. models for evaluation. It is worth noting, that in every test PaddlePaddle is compared to only one of the competitors, eg. PyTorch for SE-ResNeXt50 and TensorFlow for DeepLab V3+. Table 5 shows the results of the experiments. It is clear, that in these experiments PaddlePaddle is superior to all of the other frameworks in terms of training speed, but it is hard to evaluate the quality of the trained models, because developers did not consider the rate of convergence of the models.

In the case of PyTorch, developers provided scalability data in their talk at GTC 2019 (Fig. 7 and 8). As you can see, PyTorch shows very good scalability while increasing number of nodes. Developers also noted that switching to the c10d backend accelerates training by 19%.

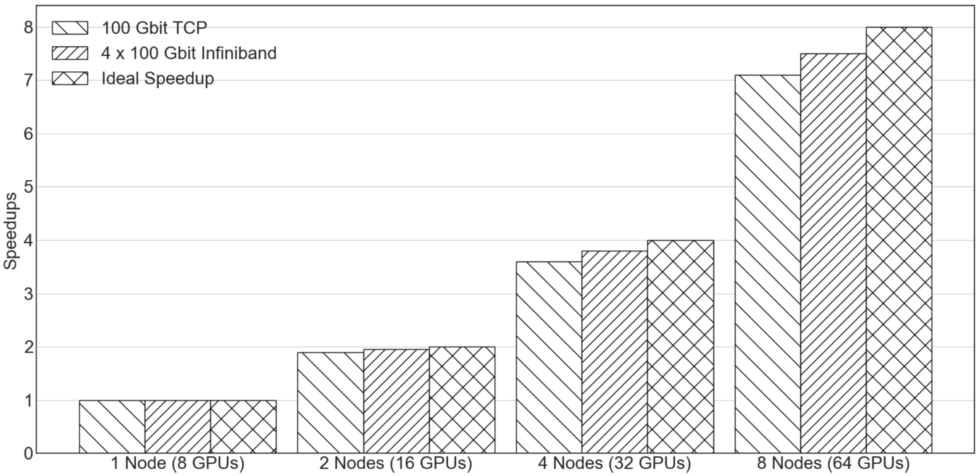


Figure 7. Pytorch 1.0 Distributed training performance on ResNet101 (NVIDIA V100)

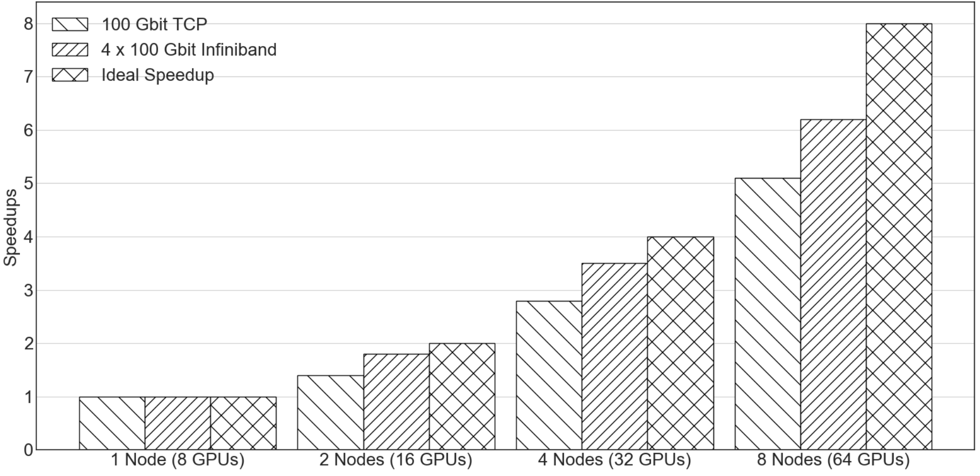


Figure 8. Pytorch 1.0 Distributed training performance on FAIR Seq (NVIDIA V100)

Conclusions

Distributed training of neural networks is becoming increasingly popular. It not only allows you to reduce the training time of a neural network, but also makes it possible to train large neural networks that cannot be fit into the memory of one machine. However, not all deep learning frameworks manage to develop distributed training quickly enough: while Multi-GPU and multithreaded training is present in all popular frameworks, the situation with distributed multi-node training is much worse. There is a good support for distributed training in the frameworks that were created with distributed training in mind: TensorFlow, MXNet and PadlePadle, as well as PyTorch and DeepLearning4J (due to integration with the Spark infrastructure).

The most popular method of distributed training is synchronous training with data parallelism. Tools for model parallelism have only recently begun to actively develop. There is Mesh TensorFlow solution; PyTorch provides the ability to manually implement model parallelism. Effective training on large supercomputers with a large number of nodes is impossible without model parallelism. We can expect that active researches will be carried out and new frameworks will appear in this area in the near future.

A large role is played not only by the performance and technical capabilities of the framework, but also by the ease of use of distributed training. This is because in order to search for a neural network that provides the necessary quality for solving the problem, it is necessary to train a large number of neural networks with different architectures and hyperparameters. Therefore, third-party libraries such as Horovod, which make distributed training using the TensorFlow, PyTorch, and MXNet frameworks easy and convenient, are gaining popularity. The TensorFlow framework is moving in the same direction, where in version 2.0 Keras has become the recommended high-level API which has integrated all distributed training tools.

You can also speed up the search for the most suitable neural network architectures using automated optimization tools for distributed hyperparameters, such as HyperOpt [48], Tune [61], Keras Tuner [27], etc. Due to the growing popularity of distributed training of neural networks, one can also expect the development of tools for distributed optimization of hyperparameters.

The process of consolidation of training frameworks for deep neural networks should not be missed out, primarily based on frameworks with the support of large Internet companies with large financial and computing resources: TensorFlow from Google and PyTorch from Facebook. The Keras framework has been included into TensorFlow; Horovod is underway to be included into the TensorFlow Distribution Strategy API [15]. PyTorch included the capabilities of the classic Torch, and it also included the Caffe2 framework, which has powerful distributed training tools. Other frameworks, unfortunately, do not develop in the field of distributed training as fast as TensorFlow and PyTorch do. It is most likely that in the future the trend of consolidating and including open projects for distributed training of neural networks and optimization of hyperparameters into TensorFlow and PyTorch will continue.

Hardware architectures specially designed for training neural networks, such as Google's Tensor Processing Unit (TPU) [8], Graphcore's Intelligence Processing Unit (IPU) [23], Nervana [21] from Intel and others, are also of interest for the development of distributed training. These architectures not only allow you to accelerate the training of neural networks, but also significantly affect the development of deep learning frameworks. In particular, the Mesh TensorFlow library was developed on the assumption that it will work on an n-dimensional grid of computing devices, which is typical for the cluster architecture on TPU [66]. The performance and quality of training a neural network in Mesh TensorFlow was evaluated in a cluster with

TPU. Although Mesh TensorFlow may work in a cluster with a different architecture, in that case performance will be lower. It is likely that over time the integration of DL frameworks and specialized equipment will become so deep and effective that it will become unprofitable to use clusters with a CPU or GPU to train neural networks.

Acknowledgements

The results described in this paper were obtained with the financial support of the grant from the Russian Federation President Fund (MK-2330.2019.9).

This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.

References

1. Apache MXNet. https://mxnet.apache.org/, accessed: 2019-12-03
2. Apache MXNet crash course. https://beta.mxnet.io/guide/crash-course/index.html, accessed: 2019-12-03
3. Caffe-MPI for deep learning. https://github.com/Caffe-MPI/Caffe-MPI.github.io, accessed: 2019-12-03
4. Caffe, Multi-GPU usage. https://github.com/BVLC/caffe/blob/master/docs/multigpu.md, accessed: 2019-12-03
5. Caffe2. https://caffe2.ai/docs, accessed: 2019-12-03
6. Caffe2, ResNet-50 training example. https://github.com/pytorch/pytorch/blob/master/caffe2/python/examples/imagenet_trainer.py, accessed: 2019-12-03
7. Caffe2, Synchronous SGD. https://caffe2.ai/docs/SynchronousSGD.html, accessed: 2019-12-03
8. Cloud TPU. https://cloud.google.com/tpu/, accessed: 2019-12-03
9. Deep learning with multiple GPUs on Rescale: Torch. https://blog.rescale.com/deep-learning-with-multiple-gpus-on-rescale-torch, accessed: 2019-12-03
10. Deeplearning4J. https://deeplearning4j.org, accessed: 2019-12-03
11. DeepLearning4J: Deep learning with Java, Spark and Power. https://www.ibm.com/developerworks/community/blogs/fe313521-2e95-46f2-817d-44a4f27eba32/entry/DeepLearning4J_Deep_Learning_with_Java_Spark_and_Power?lang=en, accessed: 2019-12-03
12. Distributed deep learning with DL4J and Spark. https://deeplearning4j.org/docs/latest/deeplearning4j-scaleout-intro, accessed: 2019-12-03

13. Distributed deep learning with Horovod and PowerAI DDL. https://developer.ibm.com/linuxonpower/2018/08/24/distributed-deep-learning-horovod-powerai-ddl/, accessed: 2019-12-03

14. Distributed training in TensorFlow. https://www.tensorflow.org/guide/distribute_strategy, accessed: 2019-12-03

15. Distribution strategy - revised API. https://github.com/tensorflow/community/blob/master/rfcs/20181016-replicator.md, accessed: 2019-12-03

16. DL4J, parallel training. https://deeplearning4j.org/tutorials/14-parallel-training, accessed: 2019-12-03

17. Getting started with Intel optimization for MXNet*. https://software.intel.com/en-us/articles/getting-started-with-intel-optimization-for-mxnet, accessed: 2019-12-03

18. Getting started with the Keras functional API. https://keras.io/getting-started/functional-api-guide/, accessed: 2019-12-03

19. Guide to multi-node training with Intel Distribution of Caffe. https://github.com/intel/caffe/wiki/Multinode-guide, accessed: 2019-12-03

20. Horovod. https://github.com/uber/horovod, accessed: 2019-12-03

21. Intel Nervana neural network processors. https://www.intel.ai/nervana-nnp, accessed: 2019-12-03

22. Intel PyTorch. https://github.com/intel/pytorch, accessed: 2019-12-03

23. Intelligence processing unit. https://www.graphcore.ai/technology, accessed: 2019-12-03

24. Intel Distribution of Caffe. https://github.com/intel/caffe, accessed: 2019-12-03

25. IntelSoftware Optimization for Torch. https://github.com/intel/torch, accessed: 2019-12-03

26. Keras: Deep learning library for MXNet, TensorFlow and Theano. https://github.com/dmlc/keras, accessed: 2019-12-03

27. Keras tuner. https://github.com/keras-team/keras-tuner, accessed: 2019-12-03

28. Meet Horovod: Ubers open source distributed deep learning framework for TensorFlow. https://eng.uber.com/horovod/, accessed: 2019-12-03

29. Microsoft cognitive toolkit. https://www.microsoft.com/en-us/cognitive-toolkit, accessed: 2019-12-03

30. Microsoft cognitive toolkit, multiple GPUs and machines. https://docs.microsoft.com/en-us/cognitive-toolkit/multiple-gpus-and-machines, accessed: 2019-12-03

31. The most popular language for machine learning and data science is ... https://www.kdnuggets.com/2017/01/most-popular-language-machine-learning-data-science.html, accessed: 2019-12-03

32. MXNet, training on multiple GPUs with gluon. https://gluon.mxnet.io/chapter07_distributed-learning/multiple-gpus-gluon.html, accessed: 2019-12-03

33. MXNet, training with multiple GPUs from scratch. https://gluon.mxnet.io/chapter07_distributed-learning/multiple-gpus-scratch.html, accessed: 2019-12-03

34. MXNet, training with multiple GPUs using model parallelism. https://mxnet.apache.org/api/faq/model_parallel_lstm, accessed: 2019-12-03

35. NVCaffe. https://docs.nvidia.com/deeplearning/frameworks/caffe-user-guide/index.html#pullnvcaffe, accessed: 2019-12-03

36. PaddlePaddle. https://github.com/PaddlePaddle/Paddle, accessed: 2019-12-03

37. PaddlePaddle benchmark. https://github.com/PaddlePaddle/benchmark, accessed: 2019-12-03

38. PaddlePaddle, manual for distributed training with fluid. https://www.paddlepaddle.org.cn/documentation/docs/en/1.5/user_guides/howto/training/cluster_howto_en.html#training-in-the-parameter-server-manner, accessed: 2019-12-03

39. PaddlePaddle, parallel executor. https://www.paddlepaddle.org.cn/documentation/docs/en/1.6/api_guides/low_level/parallel_executor_en.html, accessed: 2019-12-03

40. PaddlePaddle, single-node training. https://www.paddlepaddle.org.cn/documentation/docs/en/1.5/user_guides/howto/training/single_node_en.html, accessed: 2019-12-03

41. Protocol buffers. https://developers.google.com/protocol-buffers/, accessed: 2019-12-03

42. PyTorch. https://pytorch.org/, accessed: 2019-12-03

43. PyTorch, model parallel best practices. https://pytorch.org/tutorials/intermediate/model_parallel_tutorial.html, accessed: 2019-12-03

44. Scaling Keras model training to multiple GPUs. https://devblogs.nvidia.com/scaling-keras-training-multiple-gpus/, accessed: 2019-12-03

45. TensorFlow Roadmap. https://www.tensorflow.org/community/roadmap, accessed: 2019-12-03

46. TorchMPI. https://github.com/facebookresearch/TorchMPI, accessed: 2019-12-03

47. Abadi, M., Barham, P., Chen, J., et al.: TensorFlow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 265–283 (2016), https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

48. Bergstra, J., Yamins, D., Cox, D.D.: Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In: Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28. pp. I–115–I–123. ICML'13, JMLR.org (2013), http://dl.acm.org/citation.cfm?id=3042817.3042832

49. Cho, M., Finkler, U., Kumar, S., et al.: Powerai DDL. CoRR abs/1708.02188 (2017), http://arxiv.org/abs/1708.02188

50. Chollet, F., et al.: Keras. https://keras.io (2015)

51. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: A Matlab-like environment for machine learning. In: BigLearn, NIPS Workshop (2011)

52. Dean, J., Corrado, G.S., Monga, R., et al.: Large scale distributed deep networks. In: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1. pp. 1223–1231. NIPS'12, Curran Associates Inc., USA (2012), http://dl.acm.org/citation.cfm?id=2999134.2999271

53. Devlin, J., Chang, M., Lee, K., et al.: BERT: pre-training of deep bidirectional transformers for language understanding. CoRR abs/1810.04805 (2018), http://arxiv.org/abs/1810.04805

54. Goyal, P., Dollár, P., Girshick, R.B., et al.: Accurate, large minibatch SGD: training ImageNet in 1 hour. CoRR abs/1706.02677 (2017), http://arxiv.org/abs/1706.02677

55. He, K., Zhang, X., Ren, S., et al.: Deep residual learning for image recognition. CoRR abs/1512.03385 (2015), http://arxiv.org/abs/1512.03385

56. Huang, X., Baker, J., Reddy, R.: A historical perspective of speech recognition. Commun. ACM 57(1), 94–103 (Jan 2014), DOI: 10.1145/2500887

57. Jia, Y., Shelhamer, E., Donahue, J., et al.: Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014)

58. Johnson, M., Schuster, M., Le, Q.V., et al.: Google's multilingual neural machine translation system: Enabling zero-shot translation. CoRR abs/1611.04558 (2016), http://arxiv.org/abs/1611.04558

59. Krizhevsky, A., Sutskever, I., Hinton, et al.: ImageNet classification with deep convolutional neural networks. Commun. ACM 60(6), 84–90 (May 2017), DOI: 10.1145/3065386

60. Kurth, T., Zhang, J., Satish, N., et al.: Deep learning at 15PF: Supervised and semi-supervised classification for scientific data. CoRR abs/1708.05256 (2017), http://arxiv.org/abs/1708.05256

61. Liaw, R., Liang, E., Nishihara, R., et al.: Tune: A research platform for distributed model selection and training. CoRR abs/1807.05118 (2018), http://arxiv.org/abs/1807.05118

62. Litjens, G.J.S., Kooi, T., Bejnordi, B.E., et al.: A survey on deep learning in medical image analysis. CoRR abs/1702.05747 (2017), http://arxiv.org/abs/1702.05747

63. Liu, J., Liu, J., Dutta, J., et al.: Usability study of distributed deep learning frameworks for convolutional neural networks (2018)

64. Radford, A., Wu, J., Child, R., et al.: Language models are unsupervised multitask learners (2018), https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

65. Sergeev, A., Balso, M.D.: Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint arXiv:1802.05799 (2018)

66. Shazeer, N., Cheng, Y., Parmar, N., et al.: Mesh-TensorFlow: Deep learning for supercomputers. CoRR abs/1811.02084 (2018), http://arxiv.org/abs/1811.02084

67. Shi, S., Wang, Q., Xu, P., Chu, X.: Benchmarking state-of-the-art deep learning software tools. CoRR abs/1608.07249 (2016), http://arxiv.org/abs/1608.07249

68. Strom, N.: Scalable distributed DNN training using commodity GPU cloud computing. In: INTERSPEECH Interspeech (2015)

69. Szegedy, C., Liu, W., Jia, Y., et al.: Going deeper with convolutions. CoRR abs/1409.4842 (2014), http://arxiv.org/abs/1409.4842

70. Wang, L., Chen, Z., Liu, Y., et al.: A unified optimization approach for CNN model inference on integrated GPUs. CoRR abs/1907.02154 (2019), http://arxiv.org/abs/1907.02154

71. Xiong, W., Droppo, J., Huang, X., et al.: The Microsoft 2016 conversational speech recognition system. CoRR abs/1609.03528 (2016), http://arxiv.org/abs/1609.03528