Chainer: A Deep Learning Framework for Accelerating the Research Cycle

Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito,
Shuji Suzuki, Kota Uenishi, Brian Vogel, Hiroyuki Yamazaki Vincent
tokui@preferred.jp
Preferred Networks, Inc.
Tokyo, Japan

ABSTRACT

Software frameworks for neural networks play a key role in the development and application of deep learning methods. In this paper, we introduce the Chainer framework, which intends to provide a flexible, intuitive, and high performance means of implementing the full range of deep learning models needed by researchers and practitioners. Chainer provides acceleration using Graphics Processing Units with a familiar NumPy-like API through CuPy, supports general and dynamic models in Python through Define-by-Run, and also provides add-on packages for state-of-the-art computer vision models as well as distributed training.

CCS CONCEPTS

• **Computer systems organization → Neural networks**.

KEYWORDS

deep learning frameworks, GPU computing, distributed training, computer vision

ACM Reference Format:
Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, and Shuji Suzuki, Kota Uenishi, Brian Vogel, Hiroyuki Yamazaki Vincent. 2019. Chainer: A Deep Learning Framework for Accelerating the Research Cycle. In *The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '19), August 4–8, 2019, Anchorage, AK, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3292500.3330756

1 INTRODUCTION

Deep learning is driving the third wave of artificial intelligence research [29]. Recent investigations indicate that deep learning is moving beyond its early successes in pattern recognition and toward new applications in diverse domains and industries. To implement these research ideas, a software framework for deep learning is required.

Implementing neural networks (NNs) requires a set of specialized building blocks, including multidimensional arrays, activation functions, and automatic differentiation. To avoid duplicating these tools, many developers used open-source deep learning frameworks such as Caffe [25] or Torch [11]. Because deep learning was first used successfully in computer vision and speech recognition, early deep learning frameworks were designed primarily for feed-forward networks such as convolutional neural networks (CNNs), which are effective for analyzing fixed-length data such as images.

More recently, additional types of deep learning models have become a major research topic. Following the impressive results in game playing [36], deep reinforcement learning has become a promising research area. In addition, after recurrent neural networks (RNNs) demonstrated promising results on variable-length data such as text, the use of these models has increased. RNNs with Long Short-Term Memory (LSTM) are currently being used with success for machine translation [47] and conversation models [49].

However, as most of the existing deep learning frameworks are designed for image processing using CNNs, they lack support for abstracting data structures and training models to implement more general deep learning models. In addition, many existing frameworks use a domain-specific language for representing deep learning models, along with an interpreter to translate them into a data structure stored in memory. Therefore, developers using these frameworks cannot use standard programming language debuggers–a significant problem as debugging is a major aspect in developing and tuning deep learning models.

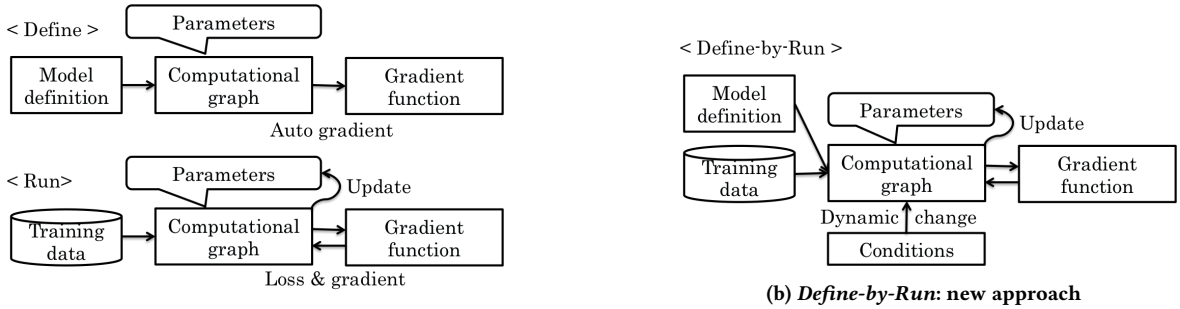
We herein introduce Chainer, an open-source framework for deep learning that provides a simple and efficient support for implementing complex algorithms, training models, and tuning model parameters. The remainder of the paper is organized as follows. Section 2 describes the standard architecture of the existing deep learning frameworks. Section 3 introduces the architecture of Chainer. Section 4 describes the performance techniques such as memory usage optimizations and double backpropagation techniques. Section 5 presents CuPy as a backend library for Graphics Processing Units (GPUs). Section 6 describes distributed training capability. Section 7 introduces ChainerCV, an add-on package for computer vision. Section 8 presents the related work. Finally, Section 9 provides a summary and the directions for future work.

2 BACKGROUND AND MOTIVATION

In typical NN frameworks, models are built in two phases, in a paradigm that we name as *Define-and-Run* (Figure 1a). In the Define phase, the computational graph of the model is first defined and constructed. This phase corresponds to the instantiation of a neural network object based on a model definition that specifies the data flow graph of inter-layer connections, initial weights, and activation functions. Automatic differentiation is typically used to define the

KDD '19, August 4–8, 2019, Anchorage, AK, USA

ACM ISBN 978-1-4503-6201-6/19/08…$15.00
https://doi.org/10.1145/3292500.3330756



(a) *Define-and-Run*: existing approach

(b) *Define-by-Run*: new approach

Figure 1: Relationship between computational graph construction and training.

computations for both the forward and backward passes, with optional graph optimizations being performed as well. In the Run phase, the actual forward and backward calculation of the graph is performed. Provided a set of training examples, the model is trained in this phase by minimizing the loss function using optimization algorithms such as stochastic gradient descent.

Under the Define-and-Run paradigm, static NN models such as CNNs can be implemented easily. The model definition may be written in a specific markup language such as Protobuf or YAML [18]. The deep learning framework serves as an interpreter that executes the model definition, which can be regarded as an independent NN program. The NN program receives the inputs (data examples), processes them (forward/backward computation), changes the model's internal state (updating), and outputs the results (predictions).

The Define-and-Run paradigm operates well for static models such as CNNs because having the full computational graph available enables potential graph optimizations to improve the memory efficiency and/or runtime performance. However, for implementing other types of NN models, two major problems arise.

The first is that it can be cumbersome to support general dynamic graphs, i.e., neural networks with control flow. In frameworks such as TensorFlow [6], the control flow decisions are defined in the data flow graph using special operators such as *Switch* and *Merge* rather than using the control flow syntaxes of the host language.

The second problem is that under the Define-and-Run paradigm, the inner mechanism of the neural network is not accessible to the user. This presents several difficulties in the creation of an effective model. For example, to debug and tune a model effectively, a user must be able to observe what is occurring inside the model. However, as a large object of a single class, the computational graph contains the entire model's information, i.e., its structure, weights, gradients, and internode operations, implying that it is essentially a black box. Consequently, development tools such as profilers and debuggers cannot determine the model's faults or how it could be improved. If graph optimizations are performed by the framework, this problem is compounded further.

3 DESIGN AND PROGRAMMING MODEL

In this section, we present the basic design of automatic differentiation APIs based on the Define-by-Run paradigm (Figure. 1b).

3.1 On Demand Graph Construction

Backpropagation is executed by bookkeeping the history of operations applied to the input arrays and backtracking the history. In the Define-by-Run paradigm, the history of operations is recorded simultaneously with the forward computation applied to concrete input arrays. This can be achieved by creating a node in the computational graph for each variable and operation. The computational graph only defines how to backtrack the operations applied to the input, and does not define the forward computation. We define the computational graph by two types of nodes: *variable nodes* that represent the variables involved in the computation, and *function nodes* that represent the operations applied to the variables. After applying a function f to the input variable x and obtaining an output variable y, the function node n_f contains a reference to the input node n_x, and the output node n_y contains a reference to the function node n_f. These references are used to backtrack the graph.

The program that defines the forward computation is similar to any standard numerical computations that do not compute any gradients. The only difference is that each differentiable function stores its computational history into the graph in addition to computing its output. Because the graph construction only relies on the execution trace of the program, it can be combined with arbitrary syntactic constructs of the host language, e.g., conditional branches and loops. Such a program generates a graph with a different topology and size at each invocation, while maintaining the correct gradient computation. The power of the host language that we can use is not limited to such primitive language constructs; we can also leverage high-level tools such as debuggers and profilers.

3.2 Object-Oriented Model Definition

Compositionality is an important characteristic of deep learning. Fragments of networks are connected in various combinations to form a rich set of architecture. APIs to write deep models should exhibit compositionality to reuse and combine components flexibly.

In the Define-by-Run paradigm, models consist of the code defining the forward computation and parameters deciding its behavior. The code is written as a host language program, and must be bound to parameters.

This parameter-binding problem is resolved by object-oriented programming. Each neural network fragment that involves its own parameters is defined by a class. Such fragments are combined into

```
class Linear(Link):
    def __init__(self, n_in, n_out):
        with self.init_scope():
            self.W = Parameter(HeNormal(),
                               (n_out, n_in))
            self.b = Parameter(0, (n_out,))

    def forward(self, x):
        return x @ self.W.T + self.b

class MultiLayerPerceptron(Chain):
    def __init__(self, n_in, n_hid, n_out):
        with self.init_scope():
            self.l1 = Linear(n_in, n_hid)
            self.l2 = Linear(n_hid, n_out)

    def forward(self, x):
        h = relu(self.l1(x))
        return self.l2(h)
```

Figure 2: Examples of model definitions by object-oriented programming.

another class to create larger model components. Hence, the modularization of neural networks and parameter binding are resolved. Figure 2 shows an example of defining a fully connected layer and a multilayer perceptron. The parameters are initialized at object construction, and the forward computation is written as a method.

Object-oriented model definition also provides a unified interface to models in terms of parameter handling. Because the model is composed of a tree of model fragments, the parameters of specific subtrees can be collected easily by traversing it.

This style of object-oriented model definition was first introduced by Chainer in 2015, and is now widely used in other Define-by-Run frameworks, e.g., PyTorch and TensorFlow Eager.

4 TECHNICAL FEATURES

In this section, we describe several techniques in Chainer that improve its simplicity and efficiency, applicable to frameworks based on the Define-by-Run paradigm.

4.1 Memory-Efficient Backpropagation

Memory efficiency is of central interest in deep learning frameworks as the sizes of models and data are limited by the amount of available physical memory. Optimization can be performed further at the framework level, especially in reducing peak memory usage.

4.1.1 Global Memory Usage Reduction. In deep learning frameworks based on the Define-by-Run paradigm, memory management is naturally delegated to that of the host language. Chainer relies on reference-counting garbage collection (GC), which is the primary mechanism for memory management in the standard Python implementation (a.k.a. CPython). Automatic differentiation APIs based on the Define-by-Run paradigm operates well with reference-counting GC; a subgraph of the computational history is released immediately once it is rendered unreachable.

When a graph is released by reference-counting GC, each node is released in the topological order of the computational graph. Meanwhile, the backpropagation algorithm visits the nodes in the topological order. By merging these two procedures, we can minimize the peak memory consumption of backpropagation. This is accomplished by manually eliminating the reference to a function node immediately after processing it during graph backtracking.

4.1.2 Function-wise Local Memory Usage Reduction. In general, the gradient (or more precisely, the Jacobian matrix) of a function depends on the input. Therefore, it is natural to design the interface of the backward computation such that it takes both the input arrays and output error as arguments. This interface, however, prevents us from applying memory usage optimization for operations that do not require the input arrays to compute the gradient. Some operations, e.g., tanh, can use the output arrays instead of the input arrays to compute the gradient. When the next operation applied to the output requires the inputs for gradient computation, we can eliminate the input arrays and retain the output arrays such that the data kept on memory for backpropagation are minimized. Further, some operations require neither the input nor the output arrays. In this case, we can eliminate the references to both of them.

Each differentiable operation is implemented as a subclass of `FunctionNode` with overridden `forward` and `backward` methods. In `forward`, the inputs and outputs required for `backward` are explicitly declared through the `retain_inputs` and `retain_outputs` method calls. If an input or output is not listed by these declarations, that input/output is not saved for backpropagation. In particular, inputs are no longer passed to the `backward` method; instead, the implementation of `backward` pulls them only when necessary.

Chainer utilizes a particular variable object representation that is designed to release memory as soon as possible once it is no longer required. In a naive implementation of automatic differentiation with the Define-by-Run paradigm, each variable node would directly contain a multidimensional array (for example, as an attribute). An issue with such a design is that the memory used by the array cannot be reclaimed until the last reference to the variable has been deleted. In particular, even if the user code does not hold any direct references to the variable, the computational graph may still hold a reference to it, in which case its memory cannot be reclaimed. This issue arises owing to the inability of distinguishing user code references from internode references. It is noteworthy that provided user code references to a variable are alive, it is necessary to maintain the multidimensional array data associated with the variable. Meanwhile, the references inside the computational graph do not always require the data to be alive; if no operation retain the variable as an input or output, the data should be released. Based on this observation, we can resolve this issue using separate objects to represent the variables in the user code and the variables in the computational graph. In Chainer, each `Variable` object holds the array data, and is distinct from the corresponding `VariableNode` object representing the variable node in the graph. The variable node object holds a reference to the array data only when the variable is retained by an operation. With this formulation, we can immediately reclaim the memory for the variable once the last reference from the user code has been removed, unless an operation retains it as an input or output.

4.2 Double Backpropagation

Backpropagating through computation involving gradient computation is a major feature of modern deep learning frameworks. It corresponds to automatic differentiation for Hessian-vector product. Such a feature is sometimes called *double backpropagation*.

Double backpropagation is supported by implementing the backward computation of each operation using functions supporting differentiation. Although this idea may appear straightforward, a naive implementation may result in reference cycles that cause unnecessary memory consumption. Two factors must be considered to avoid reference cycles: interface to access the resulting gradients, and output retention at each differentiable function.

4.2.1 Interface to Access the Resulting Gradients. Two styles of interface exist to trigger backpropagation. The first one is the `Variable.backward()` method, which computes the gradient with respect to each input. The resulting gradients are stored directly in the variable objects. The other one is the `grad()` function that takes both a set of inputs and a set of outputs as arguments. In this case, the function returns the set of computed gradients corresponding to the specified outputs. The latter interface does not introduce additional references between objects, while the former may add references from the input nodes to the computed gradients. Because the computed gradients refer the input nodes indirectly through the computational graph, a reference cycle appears.

This reference cycle is removed by discriminating between user code references and inter-node references, as discussed in Section 4.1. Because the reference from a variable to the corresponding gradient is not part of the computational graph, we place this reference into the `Variable` object instead of into the `VariableNode` object.

4.2.2 Output Retention for Double Backpropagation. As detailed in Section 4.1, the `backward` method of each `FunctionNode` implementation may use the output variables declared to be retained in the forward computation. To render backpropagation differentiable, a special step is required because the function node cannot maintain a reference to the output node; otherwise, a reference cycle is introduced. It entails that the output node may be released before the backward computation of the function node is executed. We can still maintain the validity of the differentiable backpropagation by replaying the graph construction for such an output node, i.e., a fresh node object is created and connected during backpropagation as if it were the output node. Further, we store the output array data to the function node as a backup, and use them for the recreated output node. This does not nullify the computational validity; the output node being released indicates that no other nodes or user codes contain any references to it; therefore, recreating the output node does not conflict with any existing nodes.

5 GPU SUPPORT BY CUPY

The typical usage of a deep neural network requires significant power for floating point numeric calculation; therefore, it is necessary for deep learning frameworks to fully leverage the computing power of external accelerator such as GPUs. This is not trivial for people who write deep neural network codes to implement high performance GPU programs while maintaining its flexibility, simplicity, and ease in extending components. CuPy is an open-source library

```
import numpy as np
x = np.array([1, 2])
l2 = np.linalg.norm(x)
```

(a) NumPy

```
import cupy as cp
x = cp.array([1, 2])
l2 = cp.linalg.norm(x)
```

(b) CuPy

Figure 3: Examples using NumPy and CuPy.

for Python that provides the computational power of NVIDIA GPUs with the NumPy-compatible syntax. It accelerates any computation described in a NumPy-like syntax by fully utilizing the GPU architecture with the CUDA platform provided by NVIDIA, including cuBLAS, cuDNN, cuRAND, cuSOLVER, cuSPARSE, and NCCL.

The interface of CuPy is highly compatible with that of NumPy; in most cases, it can be used as a drop-in replacement. It supports standard numerical data types, array indexing, slice, transpose, reshape, and broadcasting.

Users can create custom CUDA kernels to execute codes faster, using code snippets of C++. CuPy automatically wraps and compiles the code to create a CUDA binary. Compiled binaries are cached and reused in subsequent runs.

CuPy was first developed as the backend of Chainer. The initial version of Chainer was implemented using PyCUDA[28], a widely used Python library for CUDA GPU calculation. However, PyCUDA could not support enough functionalities of NumPy for deep learning and the CUDA support was insufficient. CuPy became independent from Chainer in June 2017, when Chainer v2.0 and CuPy v1.0 were released. Henceforth, numerous non-deep-learning projects have leveraged CuPy's strong performance and simple interface. For example, a Python-based probabilistic modeling software, Pomegranate[43], and a natural language processing library, spaCy[23], use CuPy as their GPU backend.

5.1 CuPy Example

Because CuPy is a Python package similar to NumPy, it can be imported into a Python program similarly. As shown in Fig. 3 code, `cp` is used as an abbreviation of CuPy, similar to `np` for NumPy. The `cupy.ndarray` class is in the core of CuPy as a GPU alternative of `numpy.ndarray`. In the code, `x` is an instance of `cupy.ndarray`. Its creation is identical to the NumPy syntax, except that NumPy is replaced with CuPy. The primary difference of `cupy.ndarray` from `numpy.ndarray` is that the content is allocated on the GPU memory. Most of the CuPy array manipulations are similar to those of NumPy. For example, NumPy uses `numpy.linalg.norm` to calculate the Euclidean norm on a CPU, while CuPy uses `cupy.linalg.norm` to calculate it on a GPU.

5.2 Supported Functionalities

As demonstrated in the previous subsection, CuPy implements many functions on `cupy.ndarray` objects. See the reference [1] for the supported subset of NumPy API.

5.2.1 Linear Algebra. CuPy supports most linear algebra functions in NumPy such as eigen decomposition, Cholesky decomposition, QR decomposition, singular value decomposition, linear equation

[1] https://docs-cupy.chainer.org

```
kernel = cupy.ElementwiseKernel(
    'float32 x, float32 y, float32 z',  # arguments
    'float32 w',                        # outputs
    'w = x * y + z',                    # computation
    'my_mad')                           # kernel name
w = kernel(x, y, z)
```

Figure 4: Example of a user-defined kernel.

solver, inverse of matrix, and the Moore-Penrose pseudo inverse. These functions are defined in `cupy.linalg` and are compatible with `numpy.linalg`. All of them are backed by cuSOLVER, a LAPACK implementation that operates on GPUs.

5.2.2 Sparse Matrices. CuPy supports sparse matrices using cuSPARSE. These matrices contain the same interfaces of sparse matrices in SciPy [26], `scipy.sparse`. Depending on their requirements, users can choose between coordinate-format sparse matrix, compressed sparse row matrix, compressed sparse column matrix, or sparse matrix with diagonal storage.

5.2.3 Sorting. CuPy provides sort, argsort, and lexsort functions that are compatible with NumPy, backed by Thrust, a library of parallel algorithms written in C++ with CUDA. CuPy takes the advantage of Thrust, which implements sophisticated parallel sort algorithms for GPUs.

5.3 Custom CUDA Kernels

CuPy is easy to extend with user-defined kernels by combining operators, two types of kernels, and generic types. It is easy to compose and launch an arbitrary kernel in GPUs with the CUDA code fragments.

The element-wise kernel applies the same operation to all elements. For example, the `cupy.add` function applies the + operator for each element pair. The reduction kernel folds all elements by a binary operator. For example, the `sum` function folds all elements by the + operator. Element-wise kernels and reduction kernels are analogous to Map and Reduce from MapReduce [14], respectively. Figure 4 is an example of a user-defined element-wise kernel. The first and second arguments comprise a list of input variables and a list of output variables, respectively. The definition of each variable consists of the type specifier and the name of an argument. The third argument is a CUDA code snippet that the user wants to define. In the code snippet, an arbitrary CUDA code can be used.

CuPy also supports generic types. With type parameters such as `T` specified instead of concrete types such as `float32`, custom kernels are generated as template functions. Arguments with generic types accept arbitrary types of arrays. For example, when the input type is specified as `'T x, T y'`, this function takes a pair of arrays with the same arbitrary data type such as integer and float.

6 DISTRIBUTED PARALLEL TRAINING

In this section, we introduce the distributed learning capability component of Chainer, formerly called *ChainerMN*.

Although the GPU performance has improved continuously, the training process is still time consuming even with latest GPUs. For example, training ResNet-50 [21] for the ImageNet dataset [15] typically takes as long as one week with a single GPU. Chainer's distributed capability allows integrating power of multiple GPUs fully utilizing hardware performance while preserving Chainer's flexibility enabled by its Define-by-Run approach. This allows for easy distributed learning even in complex use cases such as dynamic neural networks, generative adversarial networks, and deep reinforcement learning.

6.1 Basics of Distributed Deep Learning

6.1.1 Data and Model Parallelism. Two primary approaches are available to parallelize training by distributed processing: data parallelism and model parallelism. In data parallelism, each worker has a model replica and calculates the gradients of different minibatches. Workers update their model with these gradients collaboratively. If we define the batch size processed by each worker as b and the number of workers as n, the gradient obtained through communication is equivalent to that in the batch size bn. With more workers gradients are calculated with more training data in one iteration, thus the gradient quality is improved and accelerating the learning process.

In model parallelism, each worker has a portion of the model and cooperates with others to calculate one minibatch [13]. Model parallelism had been actively adopted particularly when GPU memory was small. Currently, data parallelism is shown to be more efficient, but in case where a model has a huge number of parameters such as domain of natural language processing, model parallelism is adopted in combination with data parallelism [45].

6.1.2 Synchronous vs. Asynchronous. Design choice on communication model from two options, synchronous or asynchronous model is the key factor to construct the overall parallel computation. Both models are described below, focusing on data parallelism.

Synchronous data parallelism in distributed training has one additional step called all-reduce step compared to non-parallelized training sequence that consists of forward computation, backward computation, and optimization. All-reduce is a parallel computing operation where the sum of parameters is calculated and distributed over all processes. This is a standard functionality of MPI. In the additional all-reduce communication step, workers communicate with each other to obtain and distribute the sum of gradients calculated by individual workers. Each worker calculates the average of gradients by dividing the sum by the number of replicas, and updates its own replica of the model with the gradient obtained through the all-reduce communication before optimization.

The asynchronous model, meanwhile, has special workers called parameter servers. The parameter server owns and controls the model parameters during the training process. Normal workers send gradients to the parameter server once the gradients are obtained by forward and backward calculations. The parameter server receives and uses the gradients to update the model. Workers receive new model parameters and start the calculation of the new gradients.

6.2 Parallelism Design

Chainer adopts data parallelism and the synchronous communication model. In the following, we will explain the choice from the options discussed in Section 6.1.

Data parallelism requires no changes but a few additions to existing implementation; splitting dataset and computing the gradient average among workers. This is because data parallelization is tantamount to increasing a minibatch size in many cases such as image recognition. Thus, we first chose the data parallelism but later added experimental support on model parallelism. Further, We adopted the synchronous communication model for its deterministic behaviour and convergence [19, 39].

Synchronous data-parallel gradient exchange can be realized by all-reduce communication between workers and thus Chainer is potentially capable of running on any communication library that supports the all-reduce operation with Chainer's communicator abstraction. The first library supported by Chainer is OpenMPI, which is especially efficient with a CUDA-aware build. However, all-reduce communication especially requires efficiency because it is called in every training iteration and needs to process a large amount of data. We adopted NCCL [5] developed by NVIDIA as a primary library to run all-reduce. NCCL is a highly-optimized communication library which enables efficient all-reduce operation between NVIDIA GPUs within and across nodes.

6.3 API Design

We describe the design goal of a distributed learning capabilty of Chainer, followed by a description of the minimal steps to extend an existing deep learning program written in Chainer to support distributed training.

The flexibility of Define-by-Run design of Chainer should not be sacrificed for distributed execution. The Define-by-Run allows the model structure to differ between iterations, only assuming that the model structures are identical between workers merely in a single iteration for all-reduce. Communication for gradient exchange occurs immediately before the optimization step, which is transparent to other Chainer components. Within the bound of this minimal assumption, any code can be put before or after the optimization step and model structure can be changed at any iteration dynamically.

Figure 5 shows a core part of a program to train the MNIST classification model, including three primary additions in distributed mode: *(1)* a communicator component that controls all inter-process communication, *(2)* transforming optimizer to `mutli_node_optimizer` to exchange gradients among workers, and *(3)* scatter the dataset to all workers.

`mutli_node_optimizer` is the most important component in making Chainer distributed. It wraps the normal optimizer and exchanges the gradient across processes using the all-reduce operation before optimizing the model. It behaves identically as the original optimizer except for the communication. At the final step, `scatter_dataset` lets workers make consensus on which fragment of training data to read for data parallelism. Training dataset are split into equal fragments and distributed over worker processes.

Requiring as minimal changes in porting to distributed mode as possible has allowed Chainer to preserve its flexibility afforded by the Define-by-Run paradigm.

```
# (1) Create a communicator
comm = chainermn.create_communicator()

# (2) Create and use multi_node_optimizer
optimizer = chainermn.create_multi_node_optimizer(
    chainer.optimizers.Adam(), comm).setup(model)

# (3) Distribute a dataset
train = chainermn.scatter_dataset(
    train, comm, shuffle=True)

# Use Chainer's Trainer class to simplify
# a forward-backward-optimization loop
iterator = chainer.iterators.SerialIterator(
    train, args.batchsize)
updater = training.StandardUpdater(
    train_iter, optimizer, device=device)
trainer = training.Trainer(updater, (100, 'epoch'))
```

Figure 5: Example of training code using ChainerMN.

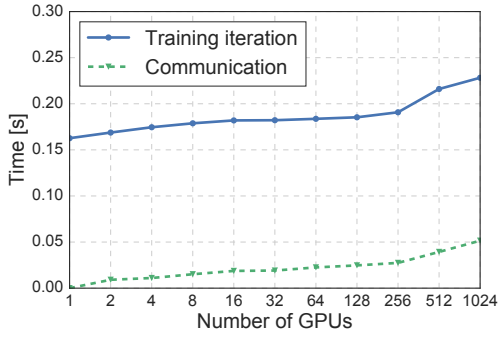


Figure 6: Iteration and communication time of ResNet-50 training for different numbers of GPUs.

6.4 Evaluation

We used the 90-epoch ResNet-50 [21] training on the ImageNet dataset as our benchmark. This task has been extensively used in evaluating the performance of distributed deep learning [10, 19, 51].

We used a cluster of 128 nodes, each of which is equipped with eight NVIDIA Tesla P100 GPUs. The per-worker minibatch size was 32 and the total minibatch size was 32k with 1024 workers. Further details of the experimental setups are provided in the appendix.

Figure 6 illustrates the communication time (i.e., all-reduce operations) and time to complete a whole iteration (i.e., forward and backward computation, communication, and optimization) for different numbers of GPUs, averaged over 100 iterations. Our scaling efficiency when using 1024 GPUs was 70% and 80% in comparison to single-GPU and single-node (i.e., 8 GPUs) baselines, respectively. Using 1024 GPUs, the mean training time over five independent runs was 897.9 ± 3.3s for 90 epochs, including the validation after each epoch.

```
# Instantiate object detection models
model_1 = FasterRCNNVGG16(pretrained_model='voc0712')
model_2 = SSD300(pretrained_model='voc0712')

# Make predictions
bboxes, labels, scores = model_1.predict([img])
bboxes, labels, scores = model_2.predict([img])
```

Figure 7: Example of executing inference with two object detection models. The two models contain the same interface to conduct an inference.

```
dataset = VOCBboxDataset(year='2007', split='test')
it = SerialIterator(dataset, batch_size=2,
    repeat=False, shuffle=False)
model = SSD300(pretrained_model='voc0712')
evaluator = DetectionVOCEvaluator(it, model,
    label_names=voc_bbox_label_names)
# Run evaluation loop
result = evaluator()
```

Figure 8: Example of performing an evaluation loop for object detection using `DetectionVOCEvaluator`.

7 CHAINERCV

In this section, we introduce *ChainerCV*, an add-on package for computer vision tasks.

Despite the powerful capability of Chainer, a gap still exists between what Chainer supports and what deep learning in computer vision requires. Deep learning models have become increasingly stronger and more complex. Therefore, it would be difficult for researchers and engineers to implement these algorithms from scratch. Additionally, many typical computer vision utilities exist, such as data loaders and pre/post-processing functions such as non-maximum suppression [12] that are outside the scope of Chainer.

ChainerCV aims at facilitating non-experts in the fast prototyping of ideas and the reduction in the barrier to enter the field. The library provides state-of-the-art models, their pre-trained weights, and training scripts for various computer vision tasks such as image classification, object detection, semantic segmentation, and instance segmentation. Additionally, the library provides utilities such as data loaders and evaluation metrics with a unified API. Our design is based on the following three principles: *easy-to-use*, *unified API*, and *reproducibility*.

7.1 Easy to use

ChainerCV supports four tasks: image classification [37], object detection [31], semantic segmentation [33], and instance segmentation [50]. For each task, several neural networks may be implemented. For instance, seven implementations are included for object detection.

Performing an inference with a ChainerCV's implementation is easy owing to a simple interface shared among models prepared for the same task. Even for models solving the same task, they differ by the output type of the neural networks and how the outputs are post-processed. The inference interface hides the difference in the underlying implementations among different models. Additionally, the inference process is simplified further by automatically downloading pre-trained weights when a model object is instantiated. Using pre-trained weights, an inference can be implemented in only two lines of the Python code, as shown in Figure 7.

In addition to supporting an easy-to-use inference, ChainerCV supports training scripts that are easily customizable for a subset of models. The training scripts are written using Chainer's training abstraction; therefore, training components can be swapped easily. The scripts are designed to be extended by users, for instance, to train with a custom user dataset.

7.2 Unified API

ChainerCV emphasizes modular design with a unified API such that users can compose the implementations in various methods. The implementations include neural network models, data loaders, evaluation metrics, and visualization utilities. The API is made consistent using the same data representation across the library. For instance, we define the data representation for images, bounding boxes, semantic pixel-wise labels, instance mask, and key points. Additionally, the API is consistent across similar functions. For example, as mentioned previously, inference methods always take an iterable of images as inputs for all models.

In addition to rendering the interface intuitive, a unified API allows us to build utilities in it. For example, we provide implementations that abstract the evaluation loop. Internally, this abstraction iterates over the dataset, performs predictions from images, and calculates the evaluation metrics using the ground truth and the predictions. It is noteworthy that the interface must be assumed for the data loader and the inference method such that the abstract utility can pass data among a data loader, an inference method, and an evaluation metric. An example is shown in Figure 8.

7.3 Reproducibility

Reproducibility in machine learning and computer vision is an important factor affecting research quality. ChainerCV aims at easing the process of reproducing the published results by providing a training code that is guaranteed to perform on par with them. These algorithms would serve as baselines to obtain a new idea through refinement and as a tool to compare a new approach against the existing approaches. With a careless implementation, the performance of trained models can easily change by deviating from the original logic and hyperparameters. This type of mistakes disqualify the implementation as a useful baseline because researchers would not be able to attain competitive results and assess the impact of their ideas properly. Table 1 shows the models supported by ChainerCV and the experimental results. The reference scores are also presented, which are reported by the original papers or the authors' implementations. As shown, the performance of our re-implementations is close to that of the original. It is noteworthy that randomness included during training can be a reason for different scores.

Table 1: Supported models in ChainerCV and their scores for (1) image classification, (2) object detection, (3) semantic segmentation, and (4) instance segmentation. For image classification, we report the top 1 error. For object detection, we report the mean average precision (mAP) for scores reported with Pascal VOC, and the average of mAP over different intersection over union threshold for scores with MS COCO. For semantic segmentation, we report the mean intersection over union (mIoU). For instance segmentation, we report the mAP of the mask.

task	dataset	model	score reference	score ours
(1)	ImageNet [37]	VGG16 [46]	28.5	29.0*
		ResNet50 [21]	24.7	24.8*
		ResNet101 [21]	23.6	23.6*
		ResNet152 [21]	23.0	23.2*
		SE-ResNet50 [24]	22.4	22.7*
		SE-ResNet101 [24]	21.8	21.8*
		SE-ResNet152 [24]	21.3	21.4*
		SE-ResNeXt50 [24]	21.0	20.9*
		SE-ResNeXt101 [24]	19.8	19.7*
(2)	Pascal VOC [17]	Faster R-CNN [42]	73.2	74.7
		SSD300 [32]	77.5	77.5
		SSD512 [32]	79.5	79.7
		YOLOv2 [40]	75.8	75.8*
		YOLOv3 [41]	80.2	80.2*
	MS COCO [31]	FPN ResNet50 [30]	36.7	37.1
		FPN ResNet101 [30]	39.4	39.5
(3)	CamVid [8]	SegNet [8]	46.3	49.4
	CityScapes [33]	PSPNet [22]	79.7	79.0*
(4)	SBD [20]	FCIS [50]	65.7	64.1

* We converted the weights of the original model

8 RELATED WORK

To the best of our knowledge, Autograd [16] had adopted the Define-by-Run paradigm to construct the backward graph before it was proposed by Chainer. Autograd is a library based on NumPy [38] and designed to enable users to write a differentiable computational graph in Python code using NumPy. However, it is not intended as a deep learning framework; therefore, it does not support GPU acceleration that is necessary for training deep models. Thus, Chainer is the first framework that focuses on deep learning workloads with the Define-by-Run paradigm. Currently, several other deep learning frameworks exist that adopt Define-by-Run. PyTorch [7] is a popular Define-by-Run framework inspired by Chainer [2], followed by Tensorflow [6] that introduced a feature called the "eager mode," which supports Define-by-Run model definitions. MXNet [9] supports imperative tensor computations which can be combined with declarative symbolic expressions by a lazy evaluation. PaddlePaddle [3] and CNTK [52] contains an imperative style as their optional programming style, while supporting the declarative style simultaneously. In contrast to most of these frameworks that support Define-by-Run optionally, Chainer is highly optimized for Define-by-Run in its overall implementation and APIs. This results in a simpler code base that reduces the cost for new developers to contribute to it.

DistBelief [13] first integrated three important techniques in the distributed training of deep neural networks, data and model parallelism, asynchronous SGD, and master-worker heterogeneous model. Subsequently, these techniques became available in Tensorflow [6], MXNet [9], and PaddlePaddle [3]. However, asynchronous SGD cannot avoid stale gradients (6.1.2) that affect accuracy and parameter server being bottleneck. To mitigate those issues, recent versions of them have added options to run synchronous SGD [39, 44].

Meanwhile, Caffe2 and PyTorch [7] use synchronous SGD. To compensate for the cost of synchronization, Caffe2 and PyTorch minimized the critical path of all-reduce communications through computation, by starting all-reduce as soon as the backward computation of a layer is completed.

Although open-sourcing models in computer vision is a widespread practice, we discovered a few studies that pursued a similar philosophy as that of ChainerCV. Primary competitors include research program codes accompanied by papers. Their primary purpose is to share research results in a verifiable manner; therefore, readability and modularity are often ignored.

Libraries more closely related to ChainerCV are *pytorch/vision* [4] and GluonCV [2]. *pytorch/vision* is a computer vision library that uses PyTorch as its backend. At the time of writing, its support for pretrained models is limited only to image classification. GluonCV is a recently released computer vision library that uses Gluon [1], which is another deep learning framework, as its backend. Similar to ChainerCV, GluonCV supports object detection, semantic segmentation, and instance segmentation. However, they do not pursue reproducibility as their core goal.

9 CONCLUSION

This paper introduced Chainer, a deep learning framework that enabled users to easily implement new algorithms and complex neural networks. Chainer has already been used successfully in a variety of leading-edge applications, including deep reinforcement learning [36], word2vec distributed representations [35], recurrent neural network language models [34], human pose estimation [48], and variational auto-encoders [27]. Because dedicated developers and users worldwide are actively collaborating on GitHub to improve Chainer, we anticipate that Chainer will become more versatile and useful in the future. In particular, the performance improvement attained by improving CuPy allows users to apply various types of deep learning models with CPU/GPU-agnostic codes. We invite all members of the deep learning community to test out Chainer and to contribute to its development.

[2] https://github.com/pytorch/pytorch/blob/v0.4.1/README.md

ACKNOWLEDGMENTS

This work could not be achieved without the help of all the contributors and the feedback from users of Chainer, CuPy, ChainerCV, and related projects. We express special thanks to them.

REFERENCES

[1] [n. d.]. Gluon. https://gluon.mxnet.io.
[2] [n. d.]. GluonCV. https://gluon-cv.mxnet.io/.
[3] [n. d.]. PaddlePaddle. http://www.paddlepaddle.org/.
[4] [n. d.]. pytorch/vision. https://github.com/pytorch/vision.
[5] 2017. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.
[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.
[7] Soumith Chintala Adam Paszke, Sam Gross and Gregory Chanan. [n. d.]. PyTorch. https://github.com/pytorch/pytorch.
[8] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. 2017. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2017).
[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274
[10] Valeriu Codreanu, Damian Podareanu, and Vikram Saletore. 2017. Achieving Deep Learning Training in less than 40 Minutes on ImageNet-1K. https://blog.surf.nl/en/imagenet-1k-training-on-intel-xeon-\phi-in-less-than-40-minutes/.
[11] R. Collobert. 2008. Torch. NIPS Workshop on Machine Learning Open Source Software.
[12] Xavier Martorell David Oro, Carles Fernandez and Javier Hernando. 2016. Work-Efficient Parallel non-maximum suppression for embedded GPU architecture. *ICASSP* (2016).
[13] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1223–1231.
[14] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters, In OSDI 2004. *OSDI '04*, 137–150.
[15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.
[16] et. al. Dougal Maclaurin. [n. d.]. Autograd. https://github.com/HIPS/autograd
[17] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. 2010. The Pascal Visual Object Classes (VOC) Challenge. *IJCV* 88, 2 (June 2010), 303–338.
[18] Ian J. Goodfellow, David Warde-Farley, Pascal Lamblin, Vincent Dumoulin, Mehdi Mirza, Razvan Pascanu, James Bergstra, Frè¿£dè¿£ric Bastien, and Yoshua Bengio. 2013. Pylearn2: a machine learning research library. *CoRR* abs/1308.4214 (2013).
[19] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR* abs/1706.02677 (2017).
[20] Bharath Hariharan, Pablo Arbelaez, Lubomir Bourdev, Subhransu Maji, and Jitendra Malik. 2011. Semantic Contours from Inverse Detectors. In *ICCV*.
[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.
[22] Xiaojuan Qi Xiaogang Wang Jiaya Jia Hengshuang Zhao, Jianping Shi. 2017. Pyramid Scene Parsing Network. *CVPR* (2017).
[23] Matthew Honnibal and Ines Montani. [n. d.]. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. ([n. d.]). https://spacy.io/
[24] Jie Hu, Li Shen, and Gang Sun. 2018. Squeeze-and-Excitation Networks. *CVPR*.
[25] Yangqing Jia. 2013. Caffe: An Open Source Convolutional Architecture for Fast Feature Embedding.
[26] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001–. SciPy: Open source scientific tools for Python. http://www.scipy.org/
[27] Diederik P. Kingma and Max Welling. 2014. Auto-Encoding Variational Bayes. *ICLR* (2014).
[28] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Comput.* 38, 3 (2012), 157–174. https://doi.org/10.1016/j.parco.2011.09.001
[29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521 (2015), 436–444.
[30] Tsung-Yi Lin, Piotr Dollár, Ross B Girshick, Kaiming He, Bharath Hariharan, and Serge J Belongie. 2017. Feature Pyramid Networks for Object Detection.. In *CVPR*, Vol. 1. 3.
[31] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *European conference on computer vision*. Springer, 740–755.
[32] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-yang Fu, and Alexander C Berg. 2016. SSD: Single Shot MultiBox Detector. *arXiv preprint arXiv:1512.02325v2* (2016).
[33] Sebastian Ramos Timo Rehfeld Markus Enzweiler Rodrigo Benenson Uwe Franke Stefan Roth Bernt Schiele Marius Cordts, Mohamed Omran. 2017. The Cityscapes Dataset for Semantic Urban Scene Understanding. *CVPR* (2017).
[34] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *INTERSPEECH*. 1045–1048.
[35] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. *NIPS* (2013), 3111–3119.
[36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. NIPS Deep Learning Workshop.
[37] Hao Su Jonathan Krause Sanjeev Satheesh Sean Ma Zhiheng Huang Andrej Karpathy Aditya Khosla Michael Bernstein Alexander C. Berg Li Fei-Fei Olga Russakovsky, Jia Deng. 2015. ImageNet Large Scale Visual Recognition Challenge. *IJCV* (2015).
[38] Travis Oliphant. 2006. *Guide to NumPy*. Trelgol Publishing. http://www.tramy.us/numpybook.pdf
[39] Xinghao Pan, Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2017. Revisiting Distributed Synchronous SGD. *ICLR Workshop Track, 2016* (02 2017).
[40] Joseph Redmon and Ali Farhadi. 2016. YOLO9000: Better, Faster, Stronger. *arXiv preprint arXiv:1612.08242* (2016).
[41] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. *arXiv* (2018).
[42] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., 91–99.
[43] Jacob Schreiber. 2017. Pomegranate: fast and flexible probabilistic modeling in python. *CoRR* abs/1711.00137 (2017).
[44] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). arXiv:1802.05799 http://arxiv.org/abs/1802.05799
[45] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Neural Information Processing Systems*.
[46] K. Simonyan and A. Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014).
[47] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. *NIPS* (2014), 3104–3112.
[48] Alexander Toshev and Christian Szegedy. 2014. DeepPose: Human Pose Estimation via Deep Neural Networks. In *CVPR*. 1653–1660.
[49] Oriol Vinyals and Quoc V. Le. 2015. A Neural Conversational Model. *CoRR* abs/1506.05869 (2015). http://dblp.uni-trier.de/db/journals/corr/corr1506.html#VinyalsL15
[50] Jifeng Dai Xiangyang Ji Yichen Wei Yi Li, Haozhi Qi. 2017. Fully Convolutional Instance-aware Semantic Segmentation. *CVPR* (2017).
[51] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2017. ImageNet Training in Minutes. *CoRR* abs/1709.05011 (2017).
[52] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Baolin Peng, and Xuedong Huang. 2014. *An Introduction to Computational Networks and the Computational Network Toolkit*. Technical Report.

A DETAILS OF EXPERIMENTAL SETUPS

We herein provides the detailed setups for experiments whose results are provided in the main text.

A.1 Distributed Training

In the experimental evaluation at Section 6.3, we used development branches based on Chainer 3.0.0rc1 and ChainerMN 1.0.0 [3]. As the underlying communication libraries, we used NCCL 2.0.5 and OpenMPI 1.10.2.

We used an in-house cluster that consists of 128 nodes. Each node is equipped with two Intel Xeon E5-2667 processors (3.20 GHz, eight cores), 256-GB memory, and eight NVIDIA Tesla P100 GPUs. All nodes are interconnected by the Mellanox Infiniband FDR.

The per-worker minibatch size was 32 and the total minibatch size was 32k with 1024 workers. Computations were generally performed in single precision; to reduce the payload size, we used half-precision floats for communication.

[3] At the time when we ran this evaluation ChainerMN was released as a plugin library to Chainer, while recently it has been merged to the mainline of Chainer.