Automated Code Transformation for Distributed Training of TensorFlow Deep Learning Models

Yusung Sim[a], Wonho Shin[a], Sungho Lee[b,*]

[a]*KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, 34141, Republic of Korea*
[b]*Chungnam National University, 99 Daehak-ro, Yuseong-gu, Daejeon, 34141, Republic of Korea*

Abstract

Distributed training of deep learning models reduces training time by parallelizing training workloads across multiple GPUs. Distributed training frameworks, such as Horovod and DeepSpeed, provide APIs, and model engineers rewrite deep learning models using the APIs to parallelize their training. However, the rewriting is time-consuming and labor-intensive because it requires engineers to read and understand documents and examples of the frameworks as well as manual efforts to rewrite code.

In this paper, we propose an automated code transformation approach that transforms TensorFlow deep learning models designed for non-distributed training to models training on multiple GPUs with the Horovod framework. We closely inspect the Horovod document and code examples and identify four common training patterns of TensorFlow deep learning models. Then, we formalize code transformation rules for each training pattern. Using the rules, we implement an automated code transformation tool that takes a TensorFlow deep learning model written in Python and rewrites it with the Horovod APIs for distributed training. Our evaluation shows that the tool correctly transforms 15 out of 16 open-source TensorFlow deep learning models. We believe that our approach significantly reduces manual efforts to parallelize training of existing TensorFlow deep learning models.

Keywords: machine learning, distributed training, code transformation, Python

*Corresponding author

1. Introduction

With recent advancements in artificial intelligence, deep learning (DL) has been widely utilized in various fields. LeCun et al. [1] define deep learning as a machine learning technique that composes multiple abstraction layers to build up a high-level representation of raw data. Model engineers construct DL models as neural networks consisting of independent layers of several perceptrons, and each layer gets input signals from the previous layer and sends output signals to the next layer. Several DL frameworks, such as TensorFlow [2] and PyTorch [3], provide APIs to easily implement deep neural networks in general purpose high-level programming languages like Python. VGG [4] and ResNet [5] for image recognition and BERT [6] and GPT-3 [7] for natural language processing are popular deep learning applications.

The deep learning development process consists of training and inference phases. The training phase improves the accuracy of DL models via multiple training steps that adjust models' parameters via forward and backward propagation. The forward propagation calculates answers of DL models for a training dataset and computes an error rate of the answers, called *loss*. Then, the backward propagation computes a model parameter gradient and adjusts parameters to reduce the loss using gradient descent algorithms. The training phase repeats the propagations multiple times over the entire training dataset for better accuracy. In the inference phase, trained DL models produce actual prediction results on input data via forward propagation.

While the training phase is essential to the DL model development, it is the most time-consuming. Model engineers train models on a huge training dataset for better accuracy, but a larger dataset also requires more training time. According to the report by You et al. [8], it takes 14 days on a single GPU to train the ResNet-50 model on the ImageNet benchmark dataset containing 1.28 million images [9]. Because models are frequently modified and retrained during development, the model development cost increases as the training time increases.

Taking advantages of parallelism, distributed training has emerged to reduce the training time. Since distributed training parallelizes training workload across multiple GPUs, model engineers can train models in significantly less time than non-distributed training. Research in various fields utilizes distributed training without losing the accuracy of trained models. Goyal et al. [10] trained the ResNet-50 model on the ImageNet benchmark in one hour with 256 GPUs, which is over 300 times faster than the non-distributed

training result. Silver et al. [11] trained AlphaGo with 176 GPUs and 48 TPUs, Zhang et al. [12] used 16 GPUs to train a speech recognition model, and Tian et al. [13] used two GPUs to train a web attack detection model on edge devices.

Meanwhile, DL models designed for non-distributed training are not directly trainable on multiple GPUs. Model engineers need to rewrite the models for distributed training with additional configurations in their training code to identify GPUs in the system, spawn processes for each GPU, and assign the training dataset to each process. There are a few distributed training frameworks, such as Horovod [14] and DeepSpeed [15], which provide APIs for distributed training without such complex configurations.

However, manually rewriting the models is error-prone, time-consuming, and labor-intensive. Even though model engineers utilize the frameworks, they should carefully read documents and code examples, and modify the training code using framework APIs accrodingly. To the best of our knowledge, there is no automated method to train models on multiple GPUs without manual modifications.

In this paper, we propose an automated approach that transforms TensorFlow DL models to ones training on multiple GPUs with the Horovod framework. We closely inspected the Horovod library documentation and the code examples that describe the code transformation required to train DL models on multiple GPUs. From the description, we identified four common training patterns used in TensorFlow DL models and the code transformation required for each training pattern. Then, we formally defined transformation rules that rewrite models with Horovod APIs for distributed training. Based on the formal rules, we implemented an automated model transformation tool for distributed training. Our tool first analyzes an input DL model to identify its training pattern and code locations on which modifications for distributed training are required. It then rewrites the model by applying the transformation rules of the identified training pattern. Our evaluation shows that our tool successfully transforms 15 out of 16 open-source TensorFlow DL models, and the transformed models with newly tuned hyperparameters train about 2.28 times faster than the original models. We also discuss the effects of distributed training of the models in the evaluation.

The contributions of this paper are as follows:

- **We formalize the code transformation rules for distributed training of TensorFlow DL models.** The formal rules allow model

engineers to understand the transformation in an explicit way rather than implicit code examples, as well as provide a basis for automation.

- **We design and implement an automated code transformation tool for distributed training.** Our tool can reduce manual efforts in rewriting models for distributed training via automation.

- **We reveal that distributed training often requires additional hyperparameter tuning.** Our empirical evaluation shows that distributed training without newly tuned hyperparameters may perform worse in training time and inference precision than non-distributed training.

2. Background

2.1. TensorFlow Deep Learning Models

This section describes two different forms of TensorFlow DL models written in Python. TensorFlow provides two major version libraries: TensorFlow 1.x published in 2016 and TensorFlow 2.x published in 2019. DL models significantly differ in their forms depending on which library they use. On TensorFlow 1.x, model engineers manually construct models as computational graphs using tensor variables and operations and execute them lazily for training and inference on an encapsulated environment called *session*. On the other hand, TensorFlow 2.x supports the eager execution that executes all tensor operations as they occur in code. With the eager execution feature, engineers no longer need to construct computational graphs and use encapsulated environments. In addition, TensorFlow 2.x integrates with the Keras library, a layer-based deep learning model library that provides a convenient interface to construct models.

Figure 1 illustrates an example neural network that classifies input images into ten categories. In the figure, vertical bars denote vectors of layers, circles denote data of vectors, and direct edges denote data dependencies from source to destination in vector operations. The network consists of three layers: an input, an output, and a hidden layer between the two layers. Each layer stores data into a vector, and the vector of a layer mutates into a vector of another layer via vector operations. In the network, the input layer has a vector of length 784, and the data in the vector is the pixels of an input image. The hidden layer is parametrized by the two-dimensional weight

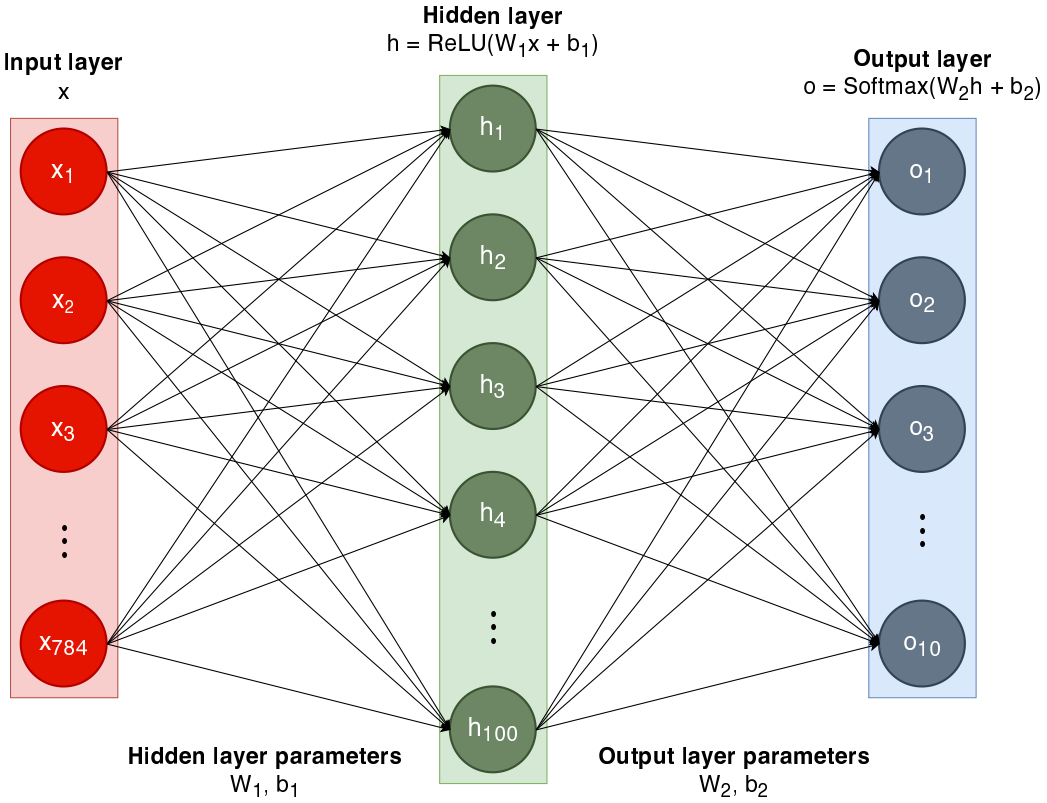


Figure 1: The neural network with a hidden layer and an output layer

matrix W_1 of size 784×100 and the bias vector b_1 of length 100. The hidden layer's vector is computed by multiplying the input layer vector x with the weight matrix W_1, adding the bias vector b_1, and finally applying the ReLU activation function that makes the results non-zero values. The output layer is parametrized by the two-dimensional weight matrix W_2 of size 100×10 and the bias vector b_2 of length 10. The output layer's vector is computed by multiplying the hidden layer vector h with the weight matrix W_2, adding the bias vector b_2, and finally applying the Softmax activation function that converts the ten data to a probability distribution of the ten categories. The weight matrices and the bias vectors in the network are model parameters, and the training phase adjusts the model parameters repeatedly, to classify input images correctly.

Figure 2 shows a TensorFlow 1.x model for the example neural network with the gradient descent training algorithm. First, lines 5 to 14 define the

```
1  import tensorflow.compat.v1 as tf
2
3  dataset = ...
4
5  x = tf.placeholder(tf.float32, [BATCH_SIZE, 784])
6  y = tf.placeholder(tf.float32, [BATCH_SIZE, 10])
7
8  W_1 = tf.Variable(tf.random_uniform([784, 100]))
9  b_1 = tf.Variable(tf.zeros([100]))
10 layer_1 = tf.nn.relu(tf.matmul(x, W_1) + b_1)
11
12 W_2 = tf.Variable(tf.random_uniform([100, 10]))
13 b_2 = tf.Variable(tf.zeros([10]))
14 layer_2 = tf.nn.softmax(tf.matmul(layer_1, W_2) + b_2)
15
16 loss = -tf.reduce_sum(y * tf.log(layer_2), 1) # Categorical
       cross entropy
17 train_op = tf.train.AdamOptimizer(0.001).minimize(loss)
18
19 with tf.Session() as sess:
20   sess.run(tf.global_variables_initializer())
21   for images, labels in dataset.take(10000):
22     sess.run(train_op, {x: images, y: labels})
```

Figure 2: TensorFlow 1.x model example

network structure and the operations between the layers. Lines 5 and 6 first create two placeholder variables, `x` and `y`, where `x` is a vector storing the pixel data of an input image, and `y` is a vector storing the answer for the classification of the input image. Lines 8 to 10 define the hidden layer. Line 8 creates a randomly initialized weight matrix `W_1`, and line 9 creates a zero-initialized bias vector `b_1`. Then, the `Variable` API wraps the matrix and the bias vector. The `Variable` API creates a model parameter of which internal values are modifiable during training. Line 10 defines the operation of the hidden layer. It first multiplies the input vector `x` with the weight matrix `W_1`, then adds the bias `b_1`, and finally applies the ReLU activation function. Note that the line does not perform the operation but only defines how the input data mutates into the hidden layer's data while executing the model. Lines 12 to 14 define the output layer. Similar to lines 8 and 9, lines 12 and 13 define a randomly-initialized weight matrix `W_2`, and a zero-initialized bias vector `b_2`, as model parameters of the output layer. Then,

line 14 defines the operation that multiplies the hidden layer output with the weight `W_2`, adds the bias vector `b_2`, and finally applies the `softmax` activation function.

After constructing the neural network, the code defines the training algorithm in lines 16 and 17. Line 16 defines the categorical cross-entropy loss function that quantifies the degree of difference between the output data of `layer_2` and the answer `y`. Line 17, then, creates an object of the `AdamOptimizer` that is an implementation of the Adam gradient descent algorithm [16]. The `minimize` method of the object generates a training operation that updates model parameters using the Adam gradient descent algorithm to minimize the loss calculated by the loss function.

Lines 19 to 22 train the model in a session. Line 19 creates a `Session` object that provides an encapsulated environment storing model parameters. Line 20 calls the `run` method of the object with a global variable initializer to initialize all the model parameters of the model in the encapsulated environment. Lines 21 and 22 then train the model in a loop that executes the training operation `train_op` with the input data `x` and `y` by iterating over the ten thousand input datasets obtained from `dataset.take(10000)`.

In TensorFlow 2.x, engineers can more easily build DL models with the Keras library and the eager execution feature. Figure 3 shows an implementation on TensorFlow 2.x for the same model shown in Figure 2. Lines 5 to 8 construct the neural network as a sequential model from a hidden layer to an output layer using the `Sequential` API of the library. The `Dense` API defines a layer by taking both the size of a vector and an activation function. The API automatically creates a weight matrix and a bias vector of the layer using the given size of a vector and the size of the vector of its previous layer. Lines 10 and 11 define a loss function and an optimizer object. Unlike TensorFlow 1.x, TensorFlow 2.x provides APIs to select a loss function used for the training, as shown in line 10. Also, while TensorFlow 1.x explicitly defines a training operation of an optimizer object and then executes it lazily on a session, TensorFlow 2.x creates an optimizer object and later uses it to execute the gradient descent algorithm eagerly during the training. Lines 13 to 19 train the model with a loop iterating over ten thousand datasets. Line 14 creates a `GradientTape` object that records executed operations for trainable model parameters within the `with` scope. Lines 15 and 16 execute the model for an input image and calculate the cross entropy loss between the execution output and the answer label. Because TensorFlow 2.x defines neural networks and loss functions as normal Python functions, they can

```
1  import tensorflow as tf
2
3  dataset = ...
4
5  model = tf.keras.Sequential([
6      tf.keras.layers.Dense(100, activation='relu'),
7      tf.keras.layers.Dense(10, activation='softmax')
8  ])
9
10 loss = tf.losses.CategoricalCrossentropy()
11 opt = tf.optimizers.Adam(0.001)
12
13 for images, labels in dataset.take(10000):
14     with tf.GradientTape() as tape:
15         probs = model(images)
16         loss_value = loss(labels, probs)
17
18     grads = tape.gradient(loss_value, model.
       trainable_variables)
19     opt.apply_gradients(zip(grads, model.trainable_variables)
       )
```

Figure 3: TensorFlow 2.x model example

be called directly with arguments, as shown in the lines. Lines 18 and 19 optimize the model parameters using the gradient descent algorithm. The `model.trainable_variable` method returns the model parameters, and the `tape.gradient` method calculates the gradients using the recorded operations on the model parameters and the loss value. Then, line 19 optimizes the model parameters according to the gradient descent algorithm by calling the `apply_gradients` method of the optimizer object.

2.2. Horovod Distributed Training Library

Horovod [14] is a Python library for distributed training of TensorFlow models. The library adopts a model-parallel approach that creates one instance of a DL model for each GPU. In the training phase, each GPU runs the computation of a single model instance for a batch of input data and computes a loss separately. The library updates the model parameters using the gradient descent algorithm with gradients computed from the average loss of the instances. Thus, engineers can take advantage of parallelism to train DL models in a shorter time on multiple GPUs.

```
1  import tensorflow.compat.v1 as tf
2  import horovod.tensorflow as hvd
3
4  hvd.init()
5  config = tf.ConfigProto()
6  config.gpu_options.allow_growth = True
7  config.gpu_options.visible_device_list = str(hvd.local_rank()
       )
8
9  dataset = ...
10
11 x = tf.placeholder(tf.float32, [BATCH_SIZE, 784])
12 y = tf.placeholder(tf.float32, [BATCH_SIZE, 10])
13
14 W_1 = tf.Variable(tf.random_uniform([784, 100]))
15 b_1 = tf.Variable(tf.zeros([100]))
16 layer_1 = tf.nn.relu(tf.matmul(x, W_1) + b_1)
17
18 W_2 = tf.Variable(tf.random_uniform([100, 10]))
19 b_2 = tf.Variable(tf.zeros([10]))
20 layer_2 = tf.nn.softmax(tf.matmul(layer_1, W_2) + b_2)
21
22 loss = -tf.reduce_sum(y * tf.log(layer_2), 1) # Categorical
       cross entropy
23 train_op = hvd.DistributedOptimizer(tf.train.AdamOptimizer
       (0.001 * hvd.size())).minimize(loss)
24
25 with tf.Session() as sess:
26   sess.run(tf.global_variables_initializer())
27   sess.run(hvd.broadcast_global_variables(root_rank=0))
28   for images, labels in dataset.take(10000 // hvd.size()):
29     sess.run(train_op, {x: images, y: labels})
```

Figure 4: Horovod distributed model example for TensorFlow 1.x model

Horovod requires model engineers to rewrite TensorFlow DL models with the Horovod API for distributed training. Figure 4 represents a distributed model rewritten from the TensorFlow 1.x model example in Figure 2. The distributed model has four big differences from the single-GPU model. 1) It configures GPUs and processes for distributed training. Line 4 initializes the Horovod configuration by calling `hvd.init()`, and lines 5 to 7 create the same number of processes with GPUs in the system and pin each GPU

per process. The GPU pinning ensures that each model instance trains on a single dedicated GPU. 2) The distributed model uses the distributed version of the gradient descent algorithm. Line 23 creates the same optimizer `tf.train.AdamOptimizer` with the learning rate multiplied by the number of GPUs, `hvd.size()`. According to the Horovod library document, the learning rate of the distributed optimizer should be scaled by the number of GPUs for efficient distributed training. Moreover, line 23 wraps the optimizer with the `DistributedOptimizer` API. The API takes a single GPU-based optimizer and produces its distributed version that averages the loss gradients across the training processes. 3) The distributed model synchronizes the model's and the optimizer's variables across the training processes. According to the Horovod library document, variables should be synchronized exactly once after initializing the variables. Therefore, line 27 broadcasts the variables across the training processes via `broadcast_global_variables` API. 4) The distributed model divides input data into multiple batches of the same number as the training processes. Lines 28 and 29 run multiple training processes simultaneously, on which each model instance gets trained with one of the batches obtained from `dataset.take(10000 // hvd.size())`.

Because TensorFlow 2.x models differ from TensorFlow 1.x, Horovod suggests different ways to rewrite models depending on the TensorFlow version. Figure 5 is a distributed model rewritten from the TensorFlow 2.x model example in Figure 3. Lines 4 and 5 initialize the Horovod configuration and a boolean flag `hvd_broadcast_done` set to `False`. Lines 7 to 11 get the list of GPUs in the system and pin each GPU per process. While TensorFlow 1.x uses `ConfigProto()` for the configuration, the lines use `config.experimental` instead since `ConfigProto()` is deprecated in TensorFlow 2.x. Line 19 defines the optimizer object with the learning rate multiplied by the number of GPUs but without wrapping `DistributedOptimizer`. Line 21 starts the training loop for each model instance with a batch of input data obtained from `dataset.take(10000 // hvd.size())`. Inside the training loop, line 26 wraps the `GradientTape` object with the Horovod API `DistributedGradientTape`. The `DistributedGradientTape` averages the loss gradients across the training processes like the `DistributedOptimizer` in the distributed TensorFlow 1.x model. Finally, lines 31 to 34 broadcast the model and optimizer variables across the training processes. TensorFlow 2.x implicitly initializes variables once when applying the optimizer's `apply_gradients` function to `model.trainable_variables`. Thus, after applying the function, the code calls the `hvd.broadcast_variables` API to

```
import tensorflow as tf
import horovod.tensorflow as hvd

hvd_broadcast_done = False
hvd.init()

gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    tf.config.experimental.set_visible_devices(gpus[hvd.
   local_rank()], 'GPU')

model = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

loss = tf.losses.CategoricalCrossentropy()
opt = tf.optimizers.Adam(0.001 * hvd.size())

for images, labels in dataset.take(10000 // hvd.size()):
    with tf.GradientTape() as tape:
        probs = model(images)
        loss_value = loss(labels, probs)

    tape = hvd.DistributedGradientTape(tape)

    grads = tape.gradient(loss_value, model.
   trainable_variables)
    opt.apply_gradients(zip(grads, model.trainable_variables)
   )

    if not hvd_broadcast_done:
        hvd.broadcast_variables(model.variables, root_rank=0)
        hvd.broadcast_variables(opt.variables(), root_rank=0)
        hvd_broadcast_done = True
```

Figure 5: Horovod distributed model example for TensorFlow 2.x model

broadcast the variables and sets the boolean flag `hvd_broadcast_done` to `True` to prevent repeated broadcasting.

3. Overview

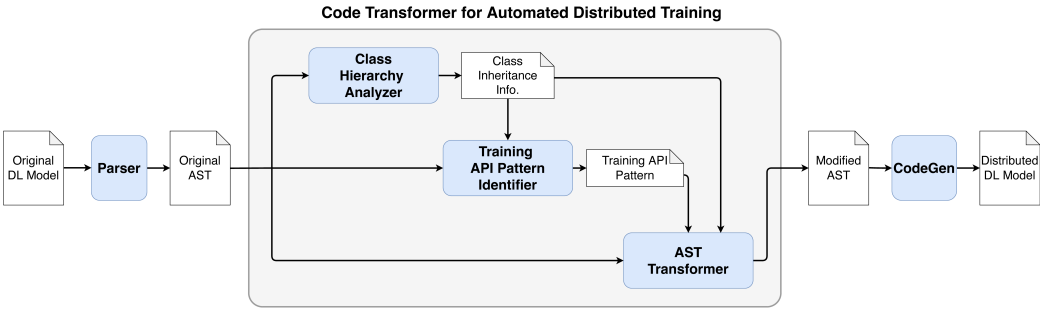


Figure 6: Overall structure of the automated transformation for distributed training

This paper proposes an automated code transformation method that rewrites TensorFlow DL models to the distributed versions with Horovod. As discussed in the previous section, distributed training with the Horovod library requires model engineers to understand the Horovod library and rewrite model code manually. To alleviate this burden, our proposed method utilizes static analysis and code transformation techniques to rewrite TensorFlow DL model code automatically based on our formal transformation rules.

Figure 6 illustrates the overview of our automated code transformation approach for distributed training of DL models with Horovod. Our approach first parses a given model into Abstract Syntax Trees (ASTs) to analyze and modify the model code mechanically. In order to define the code transformation rules for the distributed training of TensorFlow DL models, we manually inspected the Horovod library documentation and code examples. Through this analysis, we found that different transformation rules are necessary for TensorFlow models depending on the specific TensorFlow APIs used in the models. Thus, we defined four *training API patterns* that represent common code patterns of TensorFlow APIs that appear in TensorFlow DL models. The Class Hierarchy Analyzer analyzes the ASTs and extracts the class inheritance information relations between TensorFlow built-in and user-defined classes. Using the inheritance information, the Training API Pattern Analyzer identifies the training API pattern of the input model. Then, the AST Transformer selects the appropriate transformation rules based on the identified training API pattern and applies the rules to the model's ASTs. The modified ASTs are then finally converted back into a

TensorFlow DL model, and the model can now train on multiple GPUs with the support of the Horovod library.

The subsequent sections provide detailed explanations of each component of our proposed approach. Section 4 describes the necessity of the class hierarchy analysis for our approach. Section 5 explains the concept of training API patterns and the implementation of the Training API Pattern Analyzer. Finally, Section 6 provides a detailed description of the code transformation process for each identified training API pattern, including the formalization of the corresponding transformation rules.

4. Class hierarchy analysis

```
from tensorflow import keras

# `ResNet` inherits `keras.Model`
class ResNet(keras.Model):
    def __init__(self, block_list):
        ...

model = ResNet([2,2,2])
model.fit(x_train, y_train)
```

(a) Single-GPU DL model

```
from tensorflow import keras
import horovod.tensorflow.keras as hvd

class ResNet(keras.Model):
    def __init__(self, block_list):
        ...

model = ResNet([2,2,2])

model.fit(
    x_train,
    y_train,
    callbacks=[hvd.callbacks.BroadcastGlobalVariablesCallback
   (0)])
```

(b) Distributed DL model

Figure 7: Code example of distributing a single-GPU DL model using a user-defined class

Figure 7a demonstrates how ones use TensorFlow APIs to create and train models using user-defined classes. The key concept in the code is the inheritance of the TensorFlow library class `keras.Model` by the user-defined class `ResNet`, which allows the user to access the methods provided by `keras.Model` and use them to train the model. Line 4 of the code defines the user-defined class `ResNet` by inheriting from `keras.Model`. Based on the Python inheritance mechanism, the `ResNet` class inherits all the methods and attributes of the `keras.Model` class and can also define its methods and attributes. The code then creates and trains the model using the `ResNet` class instead of the `keras.Model` class. In line 8, a model is created with six blocks containing two layers each, using the `ResNet` class instead of the `keras.Model` class. Finally, in line 9, the `fit` method provided by the `keras.Model` class is called to train the network on the given data.

Distributing such models is simple, but we cannot transform them syntactically. Figure 7b demonstrates a modification of the model presented in Figure 7a to a distributed model. The example highlights the importance of recognizing the inheritance relationship between user-defined classes and the TensorFlow library classes in identifying the training-related methods. The transformation involves adding a keyword argument, `callbacks`, to the `fit` method call, as shown in line 13. However, without recognizing the inheritance relationship between `ResNet` and `keras.Model`, we cannot identify the training method call in line 9 of Figure 7a, and we cannot make the necessary modification to add the `callbacks` keyword argument.

The class hierarchy analysis is an essential pre-analysis step to solve the problem of identifying the training-related methods in DL models. The class hierarchy analysis is a static analysis technique that identifies the inheritance relationship between the classes in the code. By applying the class hierarchy analysis on the input DL models, we can identify which user-defined classes inherit TensorFlow library classes and check whether call statements target training-related methods inherited from TensorFlow library classes. In the code example in Figure 7a, the class hierarchy analyzer reads the class definition in line 4 to conclude that the class `ResNet` inherits the class `keras.Model`. The Training API Pattern Analyzer takes this information to recognize the `fit` method call as the call to the training method provided by the `keras.Model` and selects appropriate transformation rules for the training pattern. The information is also sent to AST Transformer to identify the training method call statement and apply the transformation rules.

5. Training API Pattern Identification

5.1. Training API Patterns of TensorFlow DL Models

```
for x, y in train_data.take(training_steps):
    with tf.GradientTape() as tape:
        pred = model(x, is_training=True)
        loss = loss_compute(y, pred)

    trainable_vars = model.trainable_variables
    gradients = tape.gradient(loss, trainable_vars)
    pairs = zip(gradients, trainable_vars)
    optimizer.apply_gradients(pairs)
```

(a) Using low-level training API

```
model.compile(
    optimizer = optimizer,
    loss = loss_compute)
model.fit(train_data.take(training_steps))
```

(b) Using high-level training API

Figure 8: TensorFlow model code example using two different API patterns

TensorFlow offers multiple APIs for defining the model structure and the training process. These APIs include low-level APIs such as `tf.GradientTape` and high-level APIs such as `tf.keras.Model`. The choice of API depends on the model' complexity and the project's specific requirements. Figure 8 illustrates two TensorFlow model codes that use different APIs to define the training process. Figure 8a explicitly repeats the training steps using the `for` loop and the `GradientTape` instance. On the other hand, Figure 8b uses the Keras library APIs, `compile` and `fit`, to set a training methodology of the model and invoke the training process. While both codes train the model similarly, they use different training APIs in different patterns.

Our transformation approach needs to apply different transformation rules based on the API usage in the TensorFlow model. Inspecting the Horovod documentation and open-source TensorFlow models manually, we define four categories of training API patterns that require different transformation rules. The training API patterns are the code patterns of TensorFlow API calls that commonly appear in the models belonging to the same categories. For instance, models in the GradientTape category commonly use a

`with` statement to create a `GradientTape` object, as illustrated in Figure 8a. We define patterns for such common API usages, and our tool identifies the training API patterns of given models automatically to choose appropriate transformation rules for the models.

Table 1 represents the four training API patterns. The first column shows TensorFlow versions on which models are built, and the second and third columns show training API pattern names and descriptions, respectively. We provide a detailed description of each pattern in the subsequent paragraphs and then present an algorithm that identifies the categories of models in Section 5.2.

Table 1: Four types of training API patterns

TF version	API Pattern	Description
1.x	Session	Using the `Session` API to invoke training operations
1.x	MonitoredSession	Using the `MonitoresSession` API to invoke training operations.
2.x	GradientTape	Using the `GradientTape` API to explicitly repeat the training step.
2.x	Keras	Using the `keras.Model` class to define the model and the `fit` API to train the model.

Session Pattern.. The Session pattern appears in TensorFlow 1.x models that invoke the training computation directly via the `Session` class instance. Figure 9 illustrates a code example of the Session pattern. The `with` statement in line 1 creates an instance of the `Session` class, and the `run` method called in line 3 invokes the training computation on an isolated execution environment of the session instance. The `run` method is usually called multiple times through a loop statement to train models on several training batches.

```
1 with tf.Session() as sess:
2     for images, labels in dataset.take(10000):
3         sess.run(train_op, {x: images, y: labels})
```

Figure 9: Session pattern code example

MonitoredSession Pattern.. The MonitoredSession is another typical code pattern observed in TensorFlow 1.x models. Figure 10 demonstrates a code example of the MonitoredSession pattern. This pattern bears similarities to the Session pattern, wherein the training computation is invoked directly. However, instead of utilizing an instance of the `Session` class, the MonitoredSession pattern leverages an instance of the `MonitoredSession` class that provides hooks to perform some actions automatically on specific conditions during the training. The `with` statement in line 3 creates an instance of the `MonitoredSession` class via the `MonitoredTrainingSession` API, and the `run` method call in line 5 invokes the training computation. The `SummarySaverHook` API call in line 1 creates one of the pre-defined hooks, which saves the model summaries after each training step. TensorFlow provides the `TensorBoard` utility visualizing model summaries, so that model engineers can check training processes from the summaries.

```
1 summary_hook = SummarySaverHook(...)
2
3 with MonitoredTrainingSession(hooks=[summary_hook]) as
     mon_sess:
4     while not mon_sess.should_stop():
5         mon_sess.run(train_op, feed_dict=feed_dict)
```

Figure 10: MonitoredSession pattern code example

GradientTape Pattern.. The GradientTape pattern is a classification for TensorFlow 2.x models that utilize the `GradientTape` class instance to initiate training computations manually. Figure 11 demonstrates an example of the GradientTape pattern. Line 4 initiates the `GradientTape` class instance through a `with` statement. Once creating the `GradientTape` instance, it watches all trainable variables by default and records operations executed on the variables within its context manager. The `gradient` method call in line 7 calculates and returns gradients from the recorded operations and the watched variables. Finally, in line 9, the `apply_gradients` method of the `Optimizer` class instance is called to update the model parameters.

Keras Pattern.. The Keras pattern is another classification for TensorFlow 2.x models that utilize the keras library in both model creations and training. Figure 12 represents an example of the Keras pattern. Line 1 defines the

```
optim = tf.optimizers.Adam(0.001)

for images, labels in dataset.take(10000):
    with tf.GradientTape() as tape:
        probs = model(images)
        loss_value = loss(labels, probs)
    grads = tape.gradient(loss_value, model.
    trainable_variables)
    optim.apply_gradients(zip(grads, model.
    trainable_variables))
```

Figure 11: GradientTape pattern code example

`ResNet` class inherited from the `keras.Model` class. As described in Section 4, the `keras.Model` class provides a convenient way to construct and train models. Line 5 constructs a model as an instance of the `ResNet` class. Finally, the `fit` method invoked in line 6 trains the model with the given dataset.

```
class ResNet(keras.Model):
    def __init__(self, params):
        ...

model = ResNet([2, 2, 2], num_classes)
model.fit(dataset, epochs=50)
```

Figure 12: Keras pattern code example

5.2. Training API Pattern Identifier

We implemented Training API Pattern Analyzer, which classifies a TensorFlow model into one of four training API patterns. Our approach traverses the input model AST to identify statements that match one of these patterns. Note that the input model may not contain any statements or may contain multiple statements that match the training API patterns. In such cases, the identifier must inform the user that the input model is unsuitable for the automatic transformation.

To handle these scenarios, Training API Pattern Analyzer performs a simple static analysis with a flat lattice domain composed of the four training API patterns, and *Unknown* and *Multiple* elements. The Unknown element represents that an AST does not match any of the four

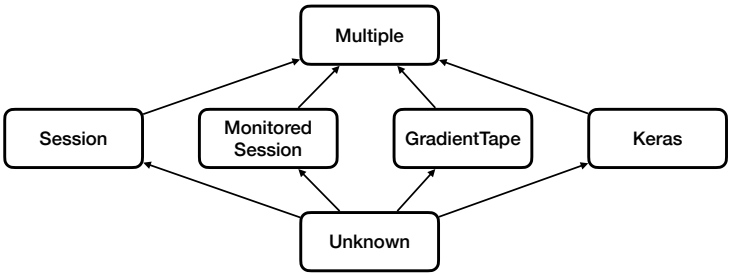


Figure 13: Flat lattice for the training API pattern identification

training API patterns, and the Multiple element represents that an AST matches multiple training API patterns. Figure 13 depicts the lattice structure. The lattice represents a partially ordered set $(P, \sqsubseteq)$ of which the boxes denote elements in P, and the directed edges denote the given order $\sqsubseteq$ between their *from* elements and their *to* elements. The order is a binary relation on the set P, which is reflexive ($\forall x \in P.\ x \sqsubseteq x$), antisymmetric ($\forall x, y \in P.\ x \sqsubseteq y \wedge x \neq y \rightarrow y \not\sqsubseteq x$), and transitive ($\forall x, y, z \in P.\ x \sqsubseteq y \wedge y \sqsubseteq z \rightarrow x \sqsubseteq z$). We also define an internal binary operation on the set P, denoted as the symbol $\sqcup$ and named *join*, which computes the least upper bound *lub* of arbitrary two elements x and y in P as follows: $x \sqsubseteq lub \wedge y \sqsubseteq lub \wedge \forall ub \in \{e \mid x \sqsubseteq e \wedge y \sqsubseteq e\}.\ lub \sqsubseteq ub$. Training API Pattern Analyzer applies the join operation to produce one identification result from a model in which the statements match multiple training API patterns. For example, the Session element results from joining Session and Unknown elements, and the Multiple element results from joining MonitoredSession and Keras elements.

Algorithm 1 describes the pseudocode of the training API pattern identification. The function `IdentifyPattern` takes both an input AST and the class hierarchy analysis result, traverses the AST to identify its training API pattern, and returns it. The algorithm first tries to match the AST with four training API patterns. The case statement in lines 3 and 4 checks whether the AST is a `with` statement that creates a `Session` instance and calls the `run` method of the instance in the body statements. If the match succeeds, the algorithm returns the Session element as the AST's identified training API pattern. The case statement in lines 5 and 6 checks whether the AST is a `with` statement that creates a `MonitoredSession` instance and calls the `run` method of the instance in the body statements. If the match succeeds, the algorithm returns the MonitoredSession element. The case

Algorithm 1 Training API pattern identification

Input: an abstract syntax tree (*ast*) and the class hierarhcy analysis result (*cha*)
Output: the identified training API pattern

```
 1: function IdentifyPattern(ast, cha)
 2:     match ast with
 3:         case with Session() as name :  body :
 4:             if body includes name.run() then Session
 5:         case with MonitoredTrainingSession() as name :  body :
 6:             if body includes name.run() then MonitoredSession
 7:         case with GradientTape() as name :  body :
 8:             if ast.parent includes name.apply_gradients() then GradientTape
 9:         case model.fit(...) :
10:             if cha.isSubclassOf(type(model), keras.Model)) then Keras
11:         otherwise :
12:             pattern ← Unknown
13:             for each child in ast.children
14:                 pattern ← pattern ⊔ IdentifyPattern(child, cha)
15:             pattern
```

statement in line 7 checks whether the AST is a `with` statement that creates a `GradientTape` instance. Line 8 also checks whether the parent AST has a child statement that calls the `apply_gradients` method of the instance. If both matches succeed, the algorithm returns the GradientTape element. The case statement in line 9 checks whether the AST is a `call` statement that invokes the `fit` method of the object assigned into an arbitrary variable *model*. The algorithm then analyzes the object's class type by tracking the variable's definition site and tests whether the class is a subclass of the `keras.Model` using the class hierarchy analysis result. If the match succeeds, the algorithm returns the Keras element. Lines 12 to 15 operate when the input AST does not match with any of the four training API patterns. The code invokes the `IdentifyPattern` function recursively for each child of the AST, joins all the results of the function calls, and returns the join operation result as the training API pattern. If the AST has no children, the algorithm identifies it as Unknown.

AST Transformer transforms an input model via appropriate transformation rules, depending on the model's training API pattern identified by Training API Pattern Analyzer. When the identified pattern is Multiple or Unknown, our tool fails to transform the model automatically and terminates. Note that the transformation failures are rare because our

evaluation shows that TRAINING API PATTERN ANALYZER identifies only one of the 16 target TensorFlow models as Multiple or Unknown.

6. Code Transformation

6.1. Formalization of Transformation Rules

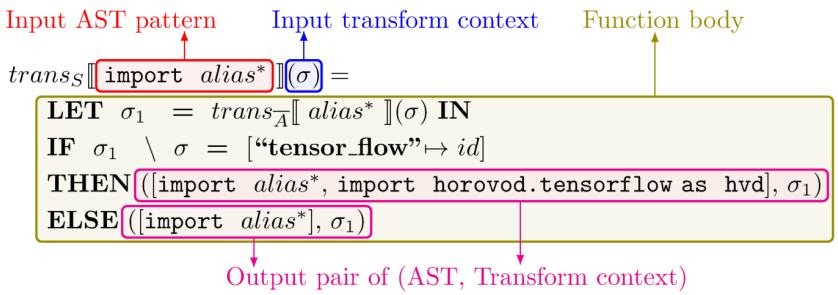


Figure 14: Transform function description

We formalize the rules for transforming single-GPU models into multi-GPU models using pure functions called *transform functions*. These functions take ASTs as input and produce ASTs as output. In addition, the functions use *transform context* objects to incorporate contextual information during the transformation process. The transform context maps strings to identifiers, which stores and propagates necessary identifiers used for the transformation. Each transform function takes a transform context as an additional input, uses and updates it in the function body, and produces the updated transform context as an additional output. This enables transform functions to utilize contextual information outside their input ASTs and pass relevant contextual information to subsequent transform functions.

We define the transform functions as a collection of partial transform functions, of which one example is illustrated in Figure 14. These partial functions operate only on ASTs that match a specified code pattern in the input AST parameter position. Any input ASTs that do not match the pattern remain unmodified. For example, the function in Figure 14 matches only `import` statements and transforms them accordingly. The function takes the transform context σ as an additional input, and its body produces a pair of the output expression and the updated transform context. We use compound expressions such as `LET-IN` and `IF-THEN-ELSE` to specify different outputs depending on certain conditions. In Figure 14, the function body

uses a `LET-IN` expression to store the updated transform context by another transform function applied to the sub phrase $alias^*$ to σ_1. Also, it uses an `IF-THEN-ELSE` expression to return different results depending on whether the `import` statement imports the `tensorflow` module or not.

```
1 import tensorflow as tf
```

(a) Original DL python code

```
1 import tensorflow as tf
2 import horovod.tensorflow as hvd
```

(b) Transformed DL python code

Figure 15: Example code transformation result

Figure 15 shows the example code transformation result by the partial transform function in Figure 14. Because the statement in Figure 15a is an import statement matched with the input AST pattern of the function, the function is responsible for transforming the statement. The function first computes the transform context σ_1 that includes all the modules and their alias identifiers imported by the statement. Next, the function calculates the difference between σ_1 and the input transform context σ. Suppose the difference contains the new entry for the string `"tensor_flow"`. In that case, the transform function returns a statement list containing the original statement and a new statement `import horovod.tensorflow as hvd` to add the import statement for the `horovod` module after the original statement, as shown in Figure 15b. Otherwise, the function returns a statement list containing only the original statement remaining unchanged.

The following subsection briefly presents essential code transform rules for each training API pattern with formal descriptions. We also provide the full transform rules as a companion report [1].

6.2. Transformation Rules for API Patterns

6.2.1. Rules for the Session Pattern

Figure 16 illustrates the required transformation of the Session pattern using two code examples, 16a the original training code and 16b its distributed training version. One of the primary transformations involves adjusting the learning rate argument in the `Optimizer` constructor call, which is achieved by multiplying it by the number of GPUs. The learning rate can be passed as

[1] https://github.com/kaist-plrg/python-analyzer/blob/main/trans/trans.pdf

```
optimizer = tf.train.MomentumOptimizer(learning_rate =
    0.01)

with tf.Session() as sess:
    for step in range(num_epochs):
        sess.run(optimizer, feed_dict)
```

(a) Session pattern example

```
optimizer = tf.train.MomentumOptimizer(learning_rate =
    0.01 * hvd.size())
optimizer = hvd.DistributedOptimizer(optimizer)

with tf.Session() as sess:
    for step in range(num_epochs):
        sess.run(optimizer, feed_dict)
```

(b) Distributed Session pattern example

Figure 16: Example transformation of the Session pattern

the first positional argument in the constructor call or as the keyword argument `learning_rate`. The other transformation uses the Horovod-provided `DistributedOptimizer` instance instead of the original `Optimizer` instance. To create the `DisributedOptimizer` instance, the original `Optimizer` is passed as an argument to its constructor.

$$
\begin{aligned}
&trans_S[\![\ id_r\ \texttt{=}\ expr_1\ \texttt{(}expr_{11}\ \ldots\ expr_{1n}\quad (id_1\ \texttt{=}\ )?\ expr_{21}\ \ldots\ (id_k\ \texttt{=}\ )?\ expr_{2k}\texttt{)}\ ]\!](\sigma) = \\
&\quad \textbf{IF}\ \ expr_1 <:_\sigma\ \ \texttt{tensorflow.keras.optimizers.Optimizer}\ \ \textbf{THEN} \\
&\qquad \textbf{IF}\ \ id_i\ \ =\ \texttt{learning\_rate}\ \ \textbf{WHEN}\ \ 1 \leq \mathrm{i} \leq \mathrm{k}\ \ \textbf{THEN} \\
&\qquad\quad ([id_r\ \texttt{=}\ expr_1\ \texttt{(}expr_{11}\ \ldots\ expr_{1n}\quad (id_1\ \texttt{=}\ )?\ expr_{21}\ \ldots\ id_i\ \texttt{=}\ expr_{2i}\ \texttt{*}\ \texttt{hvd.size()} \\
&\qquad\qquad \ldots\ (id_k\ \texttt{=}\ )?\ expr_{2k}\texttt{)},\ id_r\ \texttt{=}\ \texttt{hvd.DistributedOptimizer(}id_r\texttt{)}], \\
&\qquad\quad \sigma[\text{“}\textbf{optimizer}\text{"} \mapsto id_r]) \\
&\qquad \textbf{ELSE} \\
&\qquad\quad ([id_r\ \texttt{=}\ expr_1\ \texttt{(}expr_{11}\ \texttt{*}\ \texttt{hvd.size()}\ldots\ expr_{1n}\quad (id_1\ \texttt{=}\ )?\ expr_{21}\ \ldots\ id_i\ \texttt{=}\ expr_{2i} \\
&\qquad\qquad \ldots\ (id_k\ \texttt{=}\ )?\ expr_{2k}\texttt{)},\ id_r\ \texttt{=}\ \texttt{hvd.DistributedOptimizer(}id_r\texttt{)}], \\
&\qquad\quad \sigma[\text{“}\textbf{optimizer}\text{"} \mapsto id_r])
\end{aligned}
$$

Figure 17: Session pattern transform function: Optimizer learning rate scaling and wrapping

Figure 17 shows the essential rule for the Session pattern, which conducts the two transformations. The transform rule matches an assignment

statement that stores the result of a function call expression into a variable. Initially, the rule examines whether the callee function is the constructor of either the `Optimizer` class or any of its subclasses. The predicate $<:_\sigma$ checks the subclass relation between two classes using the class hierarchy analysis result. Then, the rule adjusts the learning rate argument of the constructor call. Suppose the learning rate is passed as a keyword argument. In that case, the rule replaces the keyword argument value with the multiplication of the original learning rate and the return value of `horovod.size()`. Otherwise, the rule adjusts the first argument instead. Following the adjustment of the learning rate, the rule adds a new statement: id_r = `hvd.DistributedOptimizer(`id_r`)`. This statement replaces the original optimizer instance with a distributed optimizer instance for any subsequent uses.

6.2.2. Rules for the MonitoredSession Pattern

Figure 18 describes an example transformation of the MonitoredSession pattern code. The `MonitoredSession` constructor optionally requires a list of `SessionRunHook` objects as a keyword argument `hooks`. To ensure consistent global variable initialization of all processes in the distributed training, the list needs to contain a `BroadcastGlobalVariableHook` object for the global variable broadcasting, as shown in Figure 18b. The instance can be created by calling the constructor of the `BroadcastGlobalVariableHook` class with the ID of a source process. When starting the training, the hook broadcasts the initial global variable parameters of the source to the other processes.

Figure 19 formalizes the transform functions for the MonitoredSession pattern. The first transform function is responsible for matching `with` statements and transforming a list of the `with_item`, each of which represents a variable and an assigned expression. The transform function denoted as $trans_{\overline{W}}$ in line 2 applies the second transform function $trans_W$ to each `with_item`. As a result, the first transform function returns a new `with` statement containing a list of the transformed `with_item`. The second transform function transforms the `with_item`. Among many forms of the `with_item`, the function matches those that assign a function call result to a variable. The second line checks whether the callee function of the `with_item` is a constructor of the subclass of the `MonitoredSession` class. If then, the transformation appends a `BroadcastGlobalVariablesHook` object to the keyword argument `hooks`.

```
with tf.train.MonitoredTrainingSession(hooks=
    hooks) as mon_sess:
     while not mon_sess.should_stop():
         mon_sess.run()
```

(a) MonitoredSession pattern example

```
with tf.train.MonitoredTrainingSession(hooks=
    hooks.append(hvd.BroadcastGlobalVariablesHook
    (0)) as mon_sess:
     while not mon_sess.should_stop():
         mon_sess.run()
```

(b) Distributed MonitoredSession pattern example

Figure 18: Example transformation of the MonitoredSession pattern

$$
\begin{aligned}
&trans_S[\![\ \texttt{with}\ \ with\_item^*\ \ :\ \ stmt^*\ ]\!](\sigma) = \\
&\quad \textbf{LET}\ \ with\_item^{*\prime}, \sigma_1 = trans_{\overline{W}}[\![\ with\_item^*\ ]\!](\sigma)\ \textbf{IN} \\
&\quad \textbf{LET}\ \ stmt^{*\prime}, \sigma_2 = trans_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma_1)\ \textbf{IN} \\
&\quad ([\texttt{with}\ \ with\_item^{*\prime}\ \ :\ \ stmt^{*\prime}], \sigma_2)
\end{aligned}
$$

$$
\begin{aligned}
&trans_W[\![\ expr_1\ \texttt{(}expr_{11}\ \ldots\ expr_{1n}\ \ \texttt{(}id_1\ \texttt{=}\ \texttt{)?}\ expr_{21}\ \ldots\ \texttt{(}id_k\ \texttt{=}\ \texttt{)?}\ expr_{2k}\texttt{)}\ \texttt{as}\ id_{as}\ ]\!](\sigma) = \\
&\quad \textbf{IF}\ \ expr_1 <:_\sigma \texttt{tensorflow.train.MonitoredSession}\ \ \textbf{THEN} \\
&\quad\quad \textbf{IF}\ \ id_i\ \ = \texttt{hooks}\ \ \textbf{WHEN}\ \ 1 \leq \mathrm{i} \leq \mathrm{k}\ \ \textbf{THEN} \\
&\quad\quad\quad (expr_1\ \texttt{(}expr_{11}\ \ldots\ expr_{1n}\ \ \texttt{(}id_1\ \texttt{=}\ \texttt{)?}\ expr_{21} \\
&\quad\quad\quad\quad \ldots\ id_i\ \texttt{=}\ expr_{2i}\texttt{.append(hvd.BroadcastGlobalVariablesHook(0))} \\
&\quad\quad\quad\quad \ldots\ \texttt{(}id_k\ \texttt{=}\ \texttt{)?}\ expr_{2k}\ \texttt{)}, \\
&\quad\quad\quad\quad \sigma[\ \textbf{“monitored\_session"}\mapsto id_{as}])
\end{aligned}
$$

Figure 19: MonitoredSession pattern transform function: Modifying `StopAtStepHook` instance

6.2.3. Rules for the GradientTape pattern

Figure 20 illustrates an example GradientTape pattern code and its distributed version. For the distributed training, the model needs to utilize a `DistributedGradientTape` instance in training instead of the `GradientTape` instance, as shown in lines 8 and 9 in Figure 20b. In addition, the model also needs to broadcast trainable global variables from the root process to the others once after applying the gradients. Lines 13 to 17 in Figure 20b represent the code that broadcasts the variables depending on the value of `hvd_broadcast_done`. Once finishing the broadcasting, the `hvd_broadcast_done`

```python
import tensorflow as tf

with tf.GradientTape() as tape:
    probs = model(images)
    loss_value = loss(labels, probs)

grads = tape.gradient(loss_value, model.
    trainable_variables)
opt.apply_gradients(zip(grads, model.
    trainable_variables))
```

(a) GradientTape pattern example

```python
import tensorflow as tf
import horovod.tensorflow as hvd
hvd_broadcast_done = False

with tf.GradientTape() as tape:
    probs = model(images)
    loss_value = loss(labels, probs)
tape = hvd.DistributedGradientTape(tape)
grads = tape.gradient(loss_value, model.
    trainable_variables)
id_new = zip(grads, model.trainable_variables)
opt.apply_gradients(id_new)

global hvd_broadcast_done
if not hvd_broadcast_done:
    hvd.broadcast_variables([x[1] for x in id_new
    ], root_rank=0,)
    hvd.broadcast_variables(opt.variables(),
    root_rank=0,)
    hvd_broadcast_done = True
```

(b) Distributed GradientTape pattern example

Figure 20: Example transformation of the GradientTape pattern

is set to `False` to prevent further broadcasting.

Figure 21 depicts the transformation for the GradientTape pattern utilizing the `DistributedGradientTape` instance. The transform function matches `with` statements and updates the transform context by transforming the list of `with_item`. The updated transform context contains a mapping from the string `graident_tape` to a variable *id* if the `with_item` creates a

$trans_S[\![$ `with` $with\_item^*$ `:` $stmt^*$ $]\!](\sigma) =$
LET $with\_item^{*\prime}, \sigma_1 = trans_{\overline{W}}[\![with\_item^*]\!](\sigma)$ **IN**
LET $stmt', \sigma_2 = trans_{\overline{S}}[\![stmt^*]\!](\sigma_1)$ **IN**
IF $\sigma_1 \setminus \sigma =$ [**"gradient_tape"** $\mapsto id$] **THEN**
([`with` $with\_item^{*\prime}$ `:` $stmt^{*\prime}$, id `= hvd.DistributedGradientTape(`id`)`], σ_2)
ELSE ([`with` $with\_item^{*\prime}$ `:` $stmt^{*\prime}$], σ_2)

Figure 21: GradientTape pattern transform function: utilizing the `DistributedGradientTape` instance

`GradientTape` instance and assigns it to the variable. Then, the function injects a new assignment statement, id = `hvd.DistributedGradientTape(id)`, right after the `with` statement, which replaces the instance of the variable with a newly created `DistributedGradientTape` instance for further uses in subsequent statements.

$trans_S[\![$ id_r `=` $expr_1$ `(`$expr_{11}$ ... $expr_{1n}$ `(`id_1 `=` `)?` $expr_{21}$ ... `(`id_k `=` `)?` $expr_{2k}$`)` $]\!](\sigma) =$
IF σ(**"optimizer"**) $= id_t$ **AND** $expr_1 = id_t$`.apply_gradients` **THEN**
LET id_z `= id_new` **IN**
([id_z `=` $expr_{11}$,
id_r `=` $expr_1$ `(`id_z $expr_{12}$ ... $expr_{1n}$ `(`id_1 `=` `)?` $expr_{21}$ ... `(`id_k `=` `)?` $expr_{2k}$`)` ,
`global hvd_braodcast_done,`
`if not hvd_broadcast_done:`
[`hvd.broadcast_variables([x[1] for x in` id_z`], root_rank=0),`
`hvd.broadcast_variables(`id_t`.variables(), root_rank=0),`
`hvd_broadcast_done = True`]
], σ)

Figure 22: GradientTape pattern transform function: broadcasting trainable global variables

Figure 22 formalizes the partial transform function that appends the variable broadcasting code after the `apply_gradients` method call. The function matches assignment statements that assign a function call result. When a variable id_t stores the `Optimizer` instance and the callee function is id_t`.apply_gradients`, the transform function changes the assignment statement and adds the variable broadcasting code. The function injects a new assignment statement that stores the first argument of the function call into a new variable id_z. The first argument of the `apply_gradients` is an iterator of tuples that contain pairs of gradients and trainable variables. After the function call statement, the transform function injects a `global` statement to refer to the global broadcast flag variable `hvd_broadcast_done` and call

statements guarded by the flag to broadcast trainable variables stored in the tuples and `opt.variables()`. The function also injects an assignment statement that sets the global broadcast flag to `True`.

6.2.4. Rules for the Keras pattern

```
class ResNet(keras.Model):
    def __init__(self, block_list):
        ...

model = ResNet([2, 2, 2])

model.fit(x_train, y_train)
```

(a) Keras pattern example

```
class ResNet(keras.Model):
    def __init__(self, block_list):
        ...

model = ResNet([2, 2, 2])

callbacks=[hvd.callbacks.
   BroadcastGlobalVariablesCallback(0)]
model.fit(x_train, y_train,
          callbacks=callbacks)
```

(b) Distributed Keras pattern example

Figure 23: Example transformation of the Keras Pattern

Figure 23 illustrates an example transformation of the Keras pattern. The Keras pattern code usually defines a subclass of `keras.Model`, such as `ResNet`, constructs a model as an instance of the class, and trains the model by calling its `fit` method. To distribute the model's training with Horovod, the `fit` method needs to take a callback `BroadcastGlobalVariablesCallback` as the keyword argument `callbacks`, shown in Figure 23b. When training starts, the callback ensures consistent initialization of all processes by broadcasting initial global variable states from a source to the other processes.

Figure 24 formalizes the transform function for the Keras pattern code. The transform function matches function call statements for the `fit` method of model objects by checking whether the receiver object of the `fit` method

$trans_S[\![\ expr_1$ `(`$expr_{11}$ ... $expr_{1n}$ `(`id_1 `=` `)`? $expr_{21}$ ... `(`id_k `=` `)`? $expr_{2k}$`)` $]\!](\sigma) =$

IF id_m `=` σ(**"model"**) **AND** $expr_1$ `=` id_m`.fit` **THEN**

IF id_i `=` `callbacks` **WHEN** $2 \leq \mathrm{i} \leq \mathrm{k}$ **THEN**

([`cb = hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0),`
`if hvd.rank() == 0: cb.append(`$expr_{2i}$`),`
$expr_1$ `(`$expr_{11}$ $\cdots$ $expr_{1n}$ `(`id_1 `=` `)`? $expr_{21}$ $\cdots$ id_i `= cb` $\cdots$ `(`id_k `=` `)`? $expr_{2k}$`)`
], σ)

ELSE

([`cb = hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0),`
$expr_1$ `(`$expr_{11}$ $\cdots$ $expr_{1n}$ `(`id_1 `=` `)`? $expr_{21}$ $\cdots$ `(`id_k `=` `)`? $expr_{2k}$
`callbacks = cb )`], σ)

Figure 24: Keras pattern transform function

call is an instance of a subclass of `keras.Model`. Then, if the function call already takes the `callbacks` keyword argument, the transform function changes the function call to three statements. The first and second statements create a temporary variable `cb` for the later use of the function call in the third statement. Note that the second statement appends the original callbacks to the `cb` variable only when the `hvd.rank()` is zero, which enables the original callbacks to be called only on the one process. If the keyword argument does not exist in the function call, the transform function propagates the callback as the `callbacks` keyword argument, shown in Figure 23b.

7. Evaluation

We evaluate the proposed approach for the following research questions:

- RQ1. (Correctness) Does the transformation correctly distribute the training of non-distributed models?
- RQ2. (Effectiveness) Does the automated transformation effectively accelerate the training of non-distributed models?

To answer the questions, we implemented an automatic code transformation tool and applied the tool to 16 TensorFlow DL models. We collected the evaluation target models from five open repositories: Hovorod [17], TensorFlow Model Garden [18], TensorFlow Examples by Americ Damien [19], CIFAR-10 Example with TensorFlow 2.0 [20], and TensorFlow 2.x Tutorials [21]. We excluded two from 18 models in the repositories because one abnormally

terminates with a runtime error in its execution, and the other duplicates. Our tool is written in Scala and is publicly available[2].

Table 2: Experiment result for the automated code transformation

Source	Model Name	API Pattern	Transform Result
TensorFlow Examples by Americ Damien [19]	LSTM-MNIST	GradientTape	◯
	SimpleCNN-GradientTape-1	GradientTape	×
Horovod GitHub [17]	SimpleCNN-GradientTape-2	GradientTape	◯
	SimpleCNN-MonitoredSession	MonitoredSession	◯
TensorFlow Model Garden [18]	SimpleCNN-Session	Session	◯
CIFAR-10 Example with TensorFlow 2.0 [20]	VGG-CIFAR10	Keras	◯
TensorFlow 2.x Tutorials [21]	Play-with-MNIST	GradientTape	◯
	Linear-Regression	GradientTape	◯
	Fashion-MNIST	Keras	◯
	CIFAR10-VGG16	GradientTape	◯
	Inception-Network	GradientTape	◯
	RNN-Sentiment-Analysis	Keras	◯
	Stacked-LSTM-ColorBot	GradientTape	◯
	Auto-Encoder	GradientTape	◯
	Variational-Auto-Encoder	GradientTape	◯
	DCGAN	GradientTape	◯

7.1. RQ1: Correctness of the transformation

To show the correctness of the transformation, we transformed each of the 16 non-distributed models using our tools and compared them to the distributed training versions of those models. For the models obtained from the Horovod repository, we used their distributed training versions available in the repository as the correct results. As for the other models, we referred to the Horovod documentation to manually transform them into distributed training versions and used those as the correct results.

Table 2 represents the transformation experiment results. The first column shows the repositories from which obtaining the models, the second column shows the model names, the third column shows the API patterns of the models, and the fourth column shows whether our tool correctly transforms the models. As shown in the table, our tool correctly transformed 15 out of the 16 models and failed to transform only the SimpleCNN-GradientTape-1 model. In the failed case, our tool raised a transformation failure error be-

[2]https://github.com/kaist-plrg/python-analyzer

```
1  # Model object is not used, instead a function used
2  def conv_net(x):
3      x = tf.reshape(x, [-1, 28, 28, 1])
4      conv1 = conv2d(x, weights['wc1'], biases['bc1'])
5      conv1 = maxpool2d(conv1, k=2)
6      conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
7      conv2 = maxpool2d(conv2, k=2)
8      fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().
       as_list()[0]])
9      fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'
       ])
10     fc1 = tf.nn.relu(fc1)
11     out = tf.add(tf.matmul(fc1, weights['out']), biases['out'
       ])
12     return tf.nn.softmax(out)
13
14 optimizer = tf.optimizers.Adam(learning_rate)
15
16 def run_optimization(x, y):
17     with tf.GradientTape() as g:
18         pred = conv_net(x)
19         loss = cross_entropy(pred, y)
20     trainable_variables = list(weights.values()) + list(
       biases.values())
21     gradients = g.gradient(loss, trainable_variables)
22     optimizer.apply_gradients(zip(gradients,
       trainable_variables))
23     # cannot perform variable broadcast with Model.variables
24
25 # training loop
26 for step, (batch_x, batch_y) in enumerate(train_data.take(
       training_steps), 1):
27     run_optimization(batch_x, batch_y)
```

Figure 25: Training code of SimpleCNN-GradientTape-1 model

cause some of the GradientTape transformation rules do not apply to the model.

We manually investigated the SimpleCNN-GradientTape-1 model code to identify the cause of the transformation failure. Figure 25 illustrates the training code snippet of the model. The code uses a GradientTape object in lines 17 to 22 to train a model constructed via a sequence of TensorFlow

Table 3: Training time comparison results

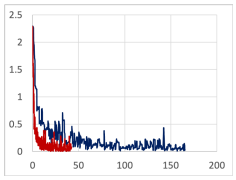
Model Name	Non-distributed training time (s)	Distributed training time (s)	Speedup
LSTM-MNIST	78.675	11.951	×6.58
SimpleCNN-GradientTape-2	3.192	1.753	×1.82
VGG-CIFAR10	967.076	299.229	×3.23
Play-with-MNIST	148.101	80.040	×1.85
Linear-Regression	0.607	0.371	×1.63
Fashion-MNIST	110.274	29.294	×3.76
CIFAR10-VGG16	1060.296	1159.293	×0.91
Inception-network	956.261	995.597	×0.96
RNN-Sentiment-Analysis	338.984	451.985	×0.74
Stacked-LSTM-ColorBot	57.327	-	-
Auto-Encoder	567.230	412.214	×1.37
Variational-Auto-Encoder	1120.291	699.777	×1.60
DCGAN	2389.052	828.428	×2.88

API calls in lines 3 to 12. As described in Section 6.2.3, our approach transforms the GradientTape pattern code by injecting statements that broadcast trainable variables of a `Model` instance. However, because the code does not construct the model as a `Model` instance, our tool cannot obtain the trainable variables from the model and fails to transform the code correctly. Note that such model construction is rare because our tool correctly transforms 11 out of the 12 GradientTape models since they utilize a `Model` instance.

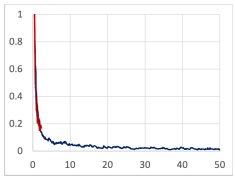
7.2. RQ2: Effectiveness of the automatically distributed training

For the second research question, we conducted a comparative analysis between the training time of the automatically transformed models and their respective original models. We calculate the training time as the time taken to reach minimum losses in training. To measure the losses, we manually injected the TensorBoard [22] API calls into the model codes, which log the loss value of each training epoch. Among the 16 models, we targeted 13 models in the experiment; we excluded two TensorFlow 1.x models with which the TensorBoard APIs are incompatible and one that our tool failed to transform. We experimented on a Linux machine with Intel Xeon CPU E5-2690 v4 @ 2.60GHz, 131GB memory, and four NVIDIA TITAN Xp GPUs.

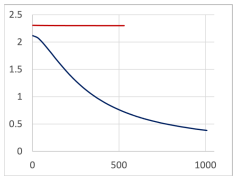
We summarize the training time comparison results in Table 3. The first column presents the model name, the second and third columns present the training time of the model, and the last column presents the training



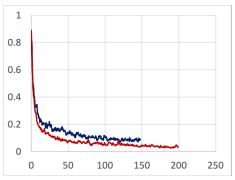
(a) LSTM-MNIST



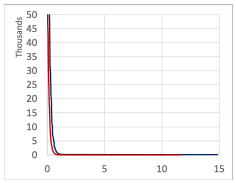
(b) SimpleCNN-GradientTape-2



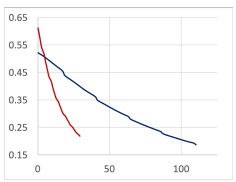
(c) VGG-CIFAR10



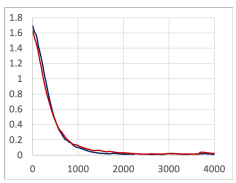
(d) Play-with-MNIST



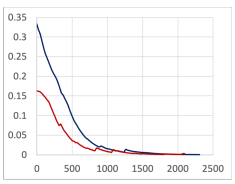
(e) Linear-Regression



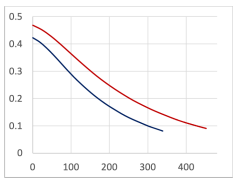
(f) Fashion-MNIST



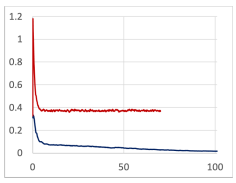
(g) CIFAR10-VGG16



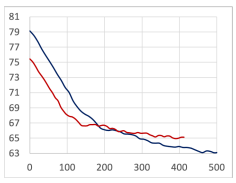
(h) Inception-network



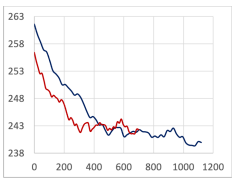
(i) RNN-Sentiment-Analysis



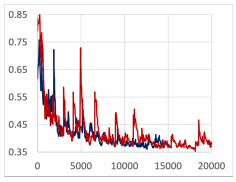
(j) Stacked-LSTM-ColorBot



(k) Auto-Encoder



(l) Variational-Auto-Encoder



(m) DCGAN

Figure 26: Comparison of training between the distributed models and the original models (X-axis: time in seconds, Y-axis: loss value, blue-line: original model, red-line: distributed model)

speed-up of the distributed model. The comparison results indicate that the training of the distributed models is, on average, approximately 2.28 times faster than that of the original models; when distributed training has a positive effect on the training time, the enhancement ranges from 1.37 times

to 6.58 times. Note that we do not record the distributed training time of the Stacked-LSTM-ColorBot model because the loss of the first distributed training epoch of the model is minimal.

Interestingly, distributed training does not ensure an acceleration of training. In three models, distributed training takes longer to reach minimum losses than non-distributed training. Furthermore, distributed training can harm the inference precision of models. Figure 26 illustrates the changes in the loss of models over training time. The x-axis represents the training time in seconds, the y-axis represents the loss values, and the blue and red lines represent the losses of the original models and their transformed models, respectively. Distributed training in six models led to higher losses than non-distributed training. For instance, in the case of the VGG-CIFAR10 model, we observed that the loss gradually decreases during the training process without distributed training. However, when adopting distributed training, the loss did not decrease significantly and remained almost unchanged, with only minor fluctuations.

Distributed training often requires additional tuning of hyperparameters to achieve positive effects. We conducted distributed training on the VGG-CIFAR10 model by applying three different learning rate parameters. Figure 27 shows the experiment result. The gray line represents the result of non-distributed training, the red line represents the result of distributed training with the original learning rate of $1e-3$, and the blue and black lines represent the results of distributed training with adjusted learning rates of $1e-4$ and $1e-5$, respectively. The three distributed training cases reached the minimum loss nearly simultaneously but exhibited significant differences in the achieved minimum loss for each training. In the case of distributed training with the original learning rate, the training led to only slight changes in loss. However, when adjusting the learning rate to $1e-4$ in distributed training, we observed that the minimum loss significantly decreased to a similar level to that of non-distributed training. In the case of distributed training with the learning rate adjusted to $1e-5$, the result for the minimum loss was better than distributed training with the original learning rate but worse than non-distributed training. This experiment shows that model engineers may need to tune hyperparameters differently from non-distributed models to obtain the benefits of distributed training.

8. Related Work

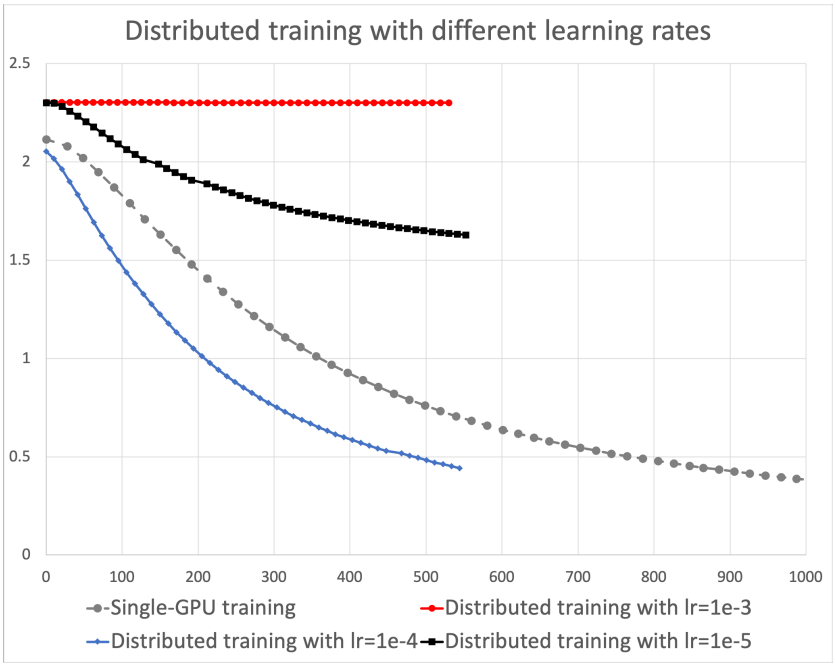


Figure 27: Distributed training on the VGG-CIFAR10 model with different learning rates

8.1. Distributed DL frameworks

Horovod [14] is a popular distributed training library that supports multiple DL frameworks such as TensorFlow and PyTorch. Besides Horovod, there are several frameworks and libraries for distributed training. TensorFlowOnSpark [23] is a Python library that combines TensorFlow with Apache Spark and Hadoop to distribute DL tasks on server clusters. DeepSpeed [15] is a distributed programming library developed by Microsoft, built on top of PyTorch [3]. DeepSpeed supports multiple distributed training methods and features, including model and pipeline parallelisms. TensorFlow officially provides a package of APIs, `tf.distribute`, for distributed training [24]. The package supports multiple implementations of `tf.distribute.Strategy`, enabling model engineers to perform distributed learning using various strategies. To train DL models in distributed environments, model engineers have to choose a library and manually rewrite the models following the library's documentation. Our approach automatically transforms existing DL models into distributed ones, reducing engineers' burdens of understanding the document and modifying models.

8.2. Code Transformation

Code transformation is techniques that modify code into a different form. Visser devised a taxonomy [25] that classifies code transformation techniques into two types: *translation*, where input and output code are written in different languages, and *rephrasing*, where input and output code are written in the same language. Our approach belongs to *renovation*, one of the subtypes of *rephrasing*, which changes the behaviors of input code and generates output code in the same language.

Researchers have proposed several code transformation techniques in Python and machine-learning domains. Loulergue and Philippe [26] devised a framework that optimizes PySke programs by automatically rewriting terms in the programs based on transformation rules they define. Haryono et al. [27] developed MLCatchUp, a code transformation tool that enables Python machine learning programs to migrate from deprecated APIs to new and stable APIs automatically. Reed et al. [28] developed `torch.fx`, a Python library for capturing and transforming PyTorch programs. Compared to these works, our work targets TensorFlow DL models written in Python and provides concrete and correct transformation rules for their distributed training.

9. Conclusion

We propose the automated approach to transform TensorFlow DL models written in Python to models training on multiple GPUs. We defined four common training patterns for TensorFlow DL models and formal code transformation rules for each pattern to parallelize training via Horovod APIs. Also, we developed a code transformation tool that takes a TensorFlow DL model, identifies its training pattern via static analysis techniques, and rewrites it for distributed training by applying transformation rules of the identified training pattern. The evaluation showed that our approach is practical in correctly transforming 15 out of 16 open-source TensorFlow DL models. We also showed that our approach is effective in that the transformed models train about 2.28 times faster than the original models. We believe that our tool reduces model engineers' burdens in rewriting models in accordance with the documentation of distributed training libraries to parallelize training.

10. Data Availability

The datasets generated and/or analyzed during the current study are available in https://github.com/kaist-plrg/python-analyzer.

Acknowledgements

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2021R1F1A1051310).

References

[1] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, nature 521 (7553) (2015) 436–444.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: a system for large-scale machine learning, in: 12th USENIX symposium on operating systems design and implementation (OSDI 16), 2016, pp. 265–283.

[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., Pytorch: An imperative style, high-performance deep learning library, Advances in neural information processing systems 32 (2019).

[4] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, arXiv preprint arXiv:1409.1556 (2014).

[5] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[6] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, arXiv preprint arXiv:1810.04805 (2018).

[7] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, Advances in neural information processing systems 33 (2020) 1877–1901.

[8] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, K. Keutzer, Imagenet training in minutes, in: Proceedings of the 47th International Conference on Parallel Processing, 2018, pp. 1–10.

[9] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al., Imagenet large scale visual recognition challenge, International journal of computer vision 115 (2015) 211–252.

[10] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, K. He, Accurate, large minibatch sgd: Training imagenet in 1 hour, arXiv preprint arXiv:1706.02677 (2017).

[11] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al., Mastering the game of go without human knowledge, nature 550 (7676) (2017) 354–359.

[12] W. Zhang, X. Cui, U. Finkler, B. Kingsbury, G. Saon, D. Kung, M. Picheny, Distributed deep learning strategies for automatic speech recognition, in: ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2019, pp. 5706–5710.

[13] Z. Tian, C. Luo, J. Qiu, X. Du, M. Guizani, A distributed deep learning system for web attack detection on edge devices, IEEE Transactions on Industrial Informatics 16 (3) (2019) 1963–1971.

[14] A. Sergeev, M. Del Balso, Horovod: fast and easy distributed deep learning in tensorflow, arXiv preprint arXiv:1802.05799 (2018).

[15] J. Rasley, S. Rajbhandari, O. Ruwase, Y. He, Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters, in: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2020, pp. 3505–3506.

[16] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).

[17] Horovod, Horovod, `https://github.com/horovod/horovod`.

[18] TensorFlow, Tensorflow model garden, `https://github.com/tensorflow/models`.

[19] A. Damien, Tensorflow examples, `https://github.com/aymericdamien/TensorFlow-Examples`.

[20] Arconsis IT-Solutions GmbH, Cifar 10 with tensorflow, `https://github.com/arconsis/cifar-10-with-tensorflow2/blob/master/BetterNetwork.py`.

[21] J. Loong, Tensorflow 2.0 tutorials, https://github.com/dragen1860/TensorFlow-2.x-Tutorials.

[22] TensorFlow, Tensorboard, https://www.tensorflow.org/tensorboard.

[23] Yahoo, Tensorflowonspark, https://github.com/yahoo/TensorFlowOnSpark.

[24] TensorFlow, Module: tf.distribute, https://www.tensorflow.org/api_docs/python/tf/distribute.

[25] E. Visser, A survey of rewriting strategies in program transformation systems, Electronic Notes in Theoretical Computer Science 57 (2001) 109–143.

[26] F. Loulergue, J. Philippe, Automatic optimization of python skeletal parallel programs, in: International Conference on Algorithms and Architectures for Parallel Processing, Springer, 2019, pp. 183–197.

[27] S. A. Haryono, F. Thung, D. Lo, J. Lawall, L. Jiang, Mlcatchup: Automated update of deprecated machine-learning apis in python, in: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2021, pp. 584–588.

[28] J. Reed, Z. DeVito, H. He, A. Ussery, J. Ansel, Torch. fx: Practical program capture and transformation for deep learning in python, Proceedings of Machine Learning and Systems 4 (2022) 638–651.