Response Letter

Thank you to the reviewers for providing constructive feedback. We have made efforts to address each one sincerely. Your feedback has greatly helped enhance the quality of the paper. Below are our actions in response to each reviewer’s comments.

Reviewers' comments to authors

Reviewer #1:

- It is difficult to judge the contributions of this work. In the first place, the paper is lengthy in various sections. I would recommend concentrating on the main details, rather than going into details for topics that are not highly relevant.

> Thank you for the comment. We revised the paper to remove unnecessary details and simplify explanations on the main details. Specifically, 1) we revised the introduction to remove unnecessary texts and highlight the motivation and contribution of our work, 2) in the background section, we simplified explanations of TensorFlow codes and Figures 1, 2 and 3, 3) we revised Section 5 to remove unnecessary details and focus explaining the examples in Figures 7 to 11, 4) we revised Section 6 to simplify explanations and focus on explaining examples in Figures 13 to 19, 5) we revised Section 7 to clarify evaluation settings and results.

- The neural network in Figure 1 is a very simple one, and spending quite a lot of space talking about the example (including the code in Figure 2) seems to be inefficient.

> Thank you for the comment. We revised the Section 2.1. by merging Figures 2 and 3, and simplifying explanations on Figures 1 to 3 to be more concise and efficient.

- It is necessary to review more related work, thereby highlighting the contributions of the proposed approach. Currently, it is not clear how the submitted work is supposed to advance state-of-the-art research.

> Thank you for the comment on the related work section. We researched and added more related work to section 8. We compared our work to previous works on automatically optimizing the distributed training of DL models to highlight that our work's primary technique is to boost training by transforming single-GPU-based training codes on source code-level. We compared our work to previous works on Python code transformation to highlight that our work is the first work on using Python code transformation technique to distribute DL training codes automatically.

- The text in the first paragraph of Page 19 describes Figure 13, however there is no association between the text and the figure. For instance, I cannot see any P, x, y, z in the figure, but these are seen in the text. Altogether, this makes the description difficult to follow.

> Thank you for the comment for the accessibility from readers. We have revised it by replacing the complex mathematical explanation with an example-based description.

- The most meaningful contribution, IMHO, is the code transformation technique presented in Section 6. Unfortunately, this section is not well written, as many technical details are poorly formulated. For example, the pseudo code in Figure 17 and Figure 19 are difficult to understand. I suggest the authors provide clear explanation to these technical details.

> Thank you for the comment. We revised Section 6 for the clarity of explanations. Specifically, the transform functions are now written in pseudocode, which is more readable and clear to the readers. In addition, we revised explanations in Section 6.2 to focus on examples given in the figures.

- Section 7 evaluates the proposed approach, using evaluation target models from five open repositories. In RQ1, the correctness of the transformation is evaluated, nevertheless it is not clear which metrics are used to judge if a transformation is correct or not.

> Thank you for the comment. We added detailed explanation on how the correctness was evaluated by the authors as following:

Precisely, the first author followed step-by-step instructions in the Horovod documentation to convert each evaluation target code into its distributed version. Subsequently, the second author independently compared them with the codes transformed by our tool. If the two codes are identical except for minor syntactic differences such as whitespaces or variable declaration order, we concluded that our tool had correctly transformed the original model code.

- The text of RQ2 says that it evaluates the effectiveness of the proposed approach. However, in the experiments, only the time is measured. I guess the authors meant actually efficiency instead of effectiveness in this research question. Please consider revising the question accordingly.

> Thank you for the suggestion. We changed “effectiveness” to “efficiency” in RQ2 and revised the corresponding explanations.

Reviewer #2:

My comments are as follows:

1. Abstract does not highlight novelty of the proposed work. It's better to add more specific details of your work.

> Thank you for the comment. We revised the abstract to more emphasize the novelty of our work. In specific, we added following sentences to highlight contribution and novelty of our work:

“By utilizing source-code level transformation, our approach allows developers to quickly scale existing DL models to multiple GPUs.

To our knowledge, our work is the first source code transformation technique that automatically distributes existing TensorFlow deep learning models at the source code level.”

2. Introduction is not focused and literature can be reorganised to strengthen literature review following contributions and discuss few relevant works.

> Thank you for the comment. We revised the introduction to focus on our main motivation. Specifically, we emphasized that the manual rewriting of DL models is an error-prone and resource-consuming process. We also added a detailed explanation of our motivation on using code transformation technique. We solve the problem of the error-prone and resource-consuming nature of the manual rewriting by modeling the human rewriting process into source code-level transformation rules and programmatically applying them to target DL model codes.

To our knowledge, our work is the first work on using source code transformation technique to automatically distribute DL models. As a result, instead of reviewing previous literature in the introduction, we focus on our motivation in the introduction and review previous literature in the Section 8 Related works.

3. Figure 1 is hard to follow, for ease of the reader its highly recommended to add more specific detail of the proposed approach.

> Thank you for the comment. As with the comment from Reviewer #1, we revised Section 2 and simplified the related explanations to remove unnecessary details.

4. Organise your results some are difficult to understand e.g., what do you want to conclude in Figure 26.

> Thank you for the comment. We revised section 7 to focus on the result and conclusion of each experiment. Specifically, 1) we revised paragraphs to mention the main result of each experiment in the starting sentences, 2) we explicitly & clearly state the conclusions from each experiment in the text, and 3) we use bold fonts to highlight the numerical statistics of the experiment results.

5. Elaborate the conclusion part in more detail and also add future directions.

> Thank you for the comment. We elaborated the conclusion part so that the main technique and the results are summarized in more detail. We also added future direction at the end of the conclusion.

Reviewer #3:

Weakness/Comments:

1. Motivation of this method is not well justified.
While the authors' effort to automate the code transformation process is appreciated, it is unclear if the method brings significant productivity improvement in practice. The ability of data-parallel training is often pre-built into deep learning frameworks (e.g., TensorFlow distributed strategy, PyTorch DDP). To data-parallel a model, user often only needs to change a few lines of code. It is not obvious if the overhead of using the proposed method (e.g., download additional libraries and dependencies, checking the correctness of the automatic transformation) is significantly smaller than reading the official documentation and making the changes manually.

> Thank you for the comment. While it is not obvious that the overhead of using our automated method is significantly smaller than manually converting the training code in a single case, we believe that our approach can reduce the burden of developers when they have to rewrite multiple, different model codes.

Our work was initially motivated by an experiment to test if the Horovod library is applicable to a diverse set of open-source TensorFlow DL models. In order to correctly transform the model codes into their distributed versions, developers have to fully understand not only the distributed training library APIs but also each model’s semantics and code structure. This process induces a large burden on developers and costs a lot of time and resources.

Our code transformation approach can reduce a significant amount of these burdens by automatically transforming most of the target model codes. While our approach does not always succeed, we showed that our approach can correctly transform most of the target codes. Thus, our approach can largely help developers to quickly distribute a large set of target DL model codes and train them on multiple GPUs.

We revised our introduction text to more emphasize the benefit of applying our approach on a large set of different DL models compared to manually rewriting each of them. Specifically, we added two paragraphs on page 3 to give details on the motivation of our work.

2. The generalizability of the method is not well discussed.
To use the method, the user code must be written in precisely one of the four patterns identified by the authors, therefore it's generalizability is very questionable. For example, as shown in the failure case, the method breaks if the user writes the code using functions instead of inheriting from the Keras Model class. How common these patterns are in real-world code is not discussed. To apply it to new frameworks (e.g., PyTorch and DeepSpeed), the patterns and transformation rules will need to be redefined, which may require significant effort. Whether it can be easily extended to other forms of parallelism (e.g., tensor and pipeline parallelism) is also not discussed.

> Thank you for the comment. It is true that our approach requires extra effort to automatically transform DL models written in other DL frameworks or distributed training libraries. To transform such models, we need to identify API usage patterns of these libraries and define corresponding transformation rules. While this requires manual effort, we only need to implement new transformation rules and reuse the code transformation infrastructure provided by our approach. However, our work limits its scope to transforming models that use only the TensorFlow library in a

data-parallel manner and therefore does not claim generality for all DL models.

3. Evaluation is not comprehensive.
The codebase used in the evaluation section is limited to simple tutorial-level training scripts. It would be better if the authors can evaluate the method on more complex codebases, e.g., code repositories for training SOTA models or those used in real-world applications.

> Thank you for the comment.

We experimented to collect and transform real-world TensorFlow models. However, it was infeasible because most publicly available real-world models are not pure TensorFlow models; they either use external training libraries or include non-Python scripts in their training process. Our approach cannot correctly transform these models as our scope is to identify and transform DL models trained with pure TensorFlow APIs.

To transform such models or applications beyond our scope, our approach should be extended as following: 1) identify common API patterns of external training libraries and define code transformation rules for the libraries, 2) extend our approach to other deep learning frameworks, such as PyTorch, 3) develop static analysis that identifies only parts related to model training from complex multilingual applications.

We added detailed explanations on how further studies can make our approach applicable to SOTA models and real-world applications in the conclusion section. In the evaluation section, we also briefly mention why we excluded real-world models from our evaluation targets.

4. Choosing training hyper-parameters.
The fact that training hyper-parameters (e.g., learning rate) needs to be adjusted for data-parallelism is an already well-known issue. While the authors mentioned that these hyper-parameters can be adjusted during the

automatic transformation, how it should be optimally adjusted is not discussed.

> Thank you for the comment. While there are several previous works on automatically tuning the hyperparameters of different training settings, there is no known method applicable to distributed training models automatically generated by our approach. In our experiments, the hyperparameters were adjusted by trial-and-error in order to optimize the training performance. We added further explanations in the last paragraph of Section 7.2 to emphasize our observation on the hyperparameters.

5. Can the correctness of the transformation be verified?
Since the method is based on automatic matching human-written patterns, it can be prone to errors. It would be better if the authors can enhance the method to provide some guarantees on the correctness of the generated code, e.g., by adding some automatic verification mechanisms.

> Thank you for the comment. Because proving the semantic equivalence of two programs is a challenging research topic, it is beyond our scope to automatically verify the correctness of our transformation. To prevent human error, the authors cross-checked the correctness of the transformed code. In addition, we will publish the transformed model codes for external researchers to check the correctness.

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. Through source-code level transformation, our approach enables developers to efficiently scale existing DL models to multiple GPUs. Our evaluation shows that the tool correctly transforms 15 out of 16 open-source TensorFlow deep learning models. To the best of our knowledge, our work is the first automatic transformation technique for distributing existing TensorFlow deep learning models at the source code level. We believe that our approach significantly reduces manual efforts to parallelize training of existing TensorFlow deep learning models.

[*]Corresponding author

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

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 modelsvia 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 modelparameter gradient and adjusts parameters to reduce the loss using gradient descent algorithms. The training phase repeats the propagations multiple times over the entire training datasetfor 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~~ The training phase is the most time-consuming stage in developing DL models. During this phase, the model's parameters are repeatedly optimized using the gradient descent algorithm over a training dataset. 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.

~~However, rewriting existing DL models for distributed training is time-consuming and labor-intensive. DL models~~ Meanwhile, DL models designed for non-distributed training are not directly trainable on multiple GPUs. Model engineers need to ~~manually~~ 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. ~~Distributed trainingframeworks,~~ To simplify distributed training, developers have introduced frameworks such as Horovod [14] and DeepSpeed [15]~~, reduce manual efforts by providing APIs for distributed training without such complex configurations~~. These frameworks provide simple APIs for defining distributed models and training them on multiple GPUs without writing complex low-level configurations.

However, manually rewriting the DL models is error-prone, time-consuming, and labor-intensive. Although distributed training frameworks can reduce the effort required to distribute DL models, developers still need to understand the frameworks' APIs and documentation thoroughly. The challenge intensifies when migrating multiple DL models simultaneously to a distributed system. To transform each model into a corresponding distributed model, developers must deeply understand not only the distributed training frameworks but also the target model's semantics and code structure. This process significantly burdens developers and increases the likelihood of errors. Consequently, manually distributing existing DL models requires significant time and resources and introduces a high risk of human errors into the codebase. ~~Nevertheless, training with the frameworks still requires additional work, such as reading documents and code examples of the frameworks , and modifying the training code using their APIs. To the best of our knowledge, there are no automated methods to train models on multiple GPUs without manual modifications.~~

~~In this paper, we propose~~ We propose to address this problem using an *automatic code transformation* technique. Our key observation is that developers follow similar steps when rewriting single-GPU-based deep learning (DL) models for distributed systems. We modeled this manual process as a source code-level transformation from single-GPU-based DL models to distributed ones. By formalizing the typical steps of this rewriting process as code transformation rules, we can programmatically apply these transformations to existing models, resulting in automatically distributed DL models. This approach effectively reduces the developer burden of manually rewriting models and lowers the cost of distributed training.

This paper presents 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**

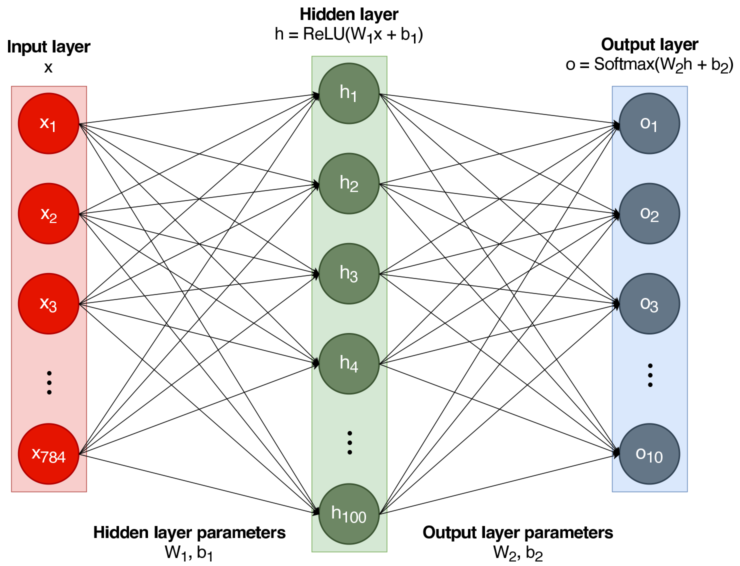


Figure 1: An example neural network

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. The network consists of three layers: an input, an output, and a hidden layer between the two layers. 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 matrix W_1 of size 784×100 and the bias vector b_1 of length 100. 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 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 **??** shows the TensorFlow 1.x training code for the neural network. First, lines 5 to 14 define the network structure and the operations between the layers. 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, and line 17 defines the Adam gradient descent algorithm [16]. Lines 19 to 22 start the main training loop by creating a `Session` object and repeatedly executing the training operation `train_op`.

~~In TensorFlow 2.x, engineers can more easily build DL models with the Keras library and the eager execution feature.~~ Figure **??** shows an implementation on TensorFlow 2.x for the same model shown in Figure ~~??~~1. 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 DenseAPI 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~~ by iterating a loop over ten thousand datasets. ~~Line 14 creates a~~ The code uses the `GradientTape` ~~object that records executed operations for trainable model parameters within the withscope. Lines 15~~ API to perform forward- 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 be called directly with~~

~~arguments, as shown in the lines.~~ back-propagations. The model parameters are optimized based on the chosen gradient descent algorithm. ~~Lines 18 and 19 optimize the model parameters using the gradient descent algorithm. The model.trainable_variablemethod returns the model parameters, and the tape.gradientmethod 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_gradientsmethod 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~~ copies the same 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~~ and distributes multiple batches of training data across the GPUs. This allows engineers to harness parallelism to train DL models on multiple GPUs in a shorter time~~on multiple GPUs~~.

Horovod requires model engineers to rewrite TensorFlow DL models with the Horovod API for distributed training. Figure 3 represents a distributed model rewritten from the TensorFlow 1.x model example in Figure ??. The distributed model has four big differences from the single-GPU model. 1) It configures GPUs and processes for distributed training. ~~Line~~ Lines 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~~ a Horovod configuration and pin each GPU ~~per process. The GPU pinning ensures that each model instance trains on~~ with a single dedicated ~~GPU.~~ model instance. 2) The distributed model uses the distributed version of the gradient descent algorithm. Line 23 ~~creates the same optimizer~~ multiplies the learning rate of the optimizer object by ~~`tf.train.AdamOptimizerwith 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~~ , and wraps it 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)

```
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 3: Horovod distributed model example for TensorFlow 1.x model

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~~ via the `broadcast_global_variables` API ~~.~~ in line 27. 4) The distributed model divides input data into multiple batches of the same number as the training processes~~. Lines~~ , as shown in lines 28 and ~~29 run multiple training~~

```python
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 4: Horovod distributed model example for TensorFlow 2.x model

~~processes simultaneously, on which each model instance gets trained with one of the batches obtained from dataset.take(10000 // hvd.size()).~~ 29.

~~Because TensorFlow 2.x models differ from TensorFlow 1.x, Horovod~~

~~suggests different ways to rewrite models depending on the TensorFlow version.~~ Figure 4 is a distributed model rewritten from the TensorFlow 2.x model example in Figure ??. ~~Lines 4 and 5 initialize the Horovod configuration and a boolean flag hvd_broadcast_doneset to False. Lines 7 to 11 get the list of GPUs in the system and pin each GPU per process. While TensorFlow~~ There are two main differences in distributing TensorFlow 2.x models compared to TensorFlow 1.x ~~uses ConfigProto()for the configuration, the lines use config.experimentalinstead since ConfigProto()is deprecated in~~ models. First, the main training loop is eagerly executed in the TensorFlow 2.x ~~. Line 19 defines the optimizer object with the learning rate multiplied by the number of GPUs but without wrapping~~ version; thus the Horovod API is changed accordingly. For instance, the ~~`DistributedOptimizer`. Line 21 starts the training loop for each model instancewith a batch of input data obtained from dataset.take(10000 // hvd.size())~~`DistributedGradientTape` ~~. Inside the training loop,~~ API in line 26 ~~wraps~~ supports the automatic gradient computation in TensorFlow 2.x. Second, the ~~GradientTapeobject with the Horovod API DistributedGradientTape. The DistributedGradientTapeaverages the loss gradients across the training processes like the DistributedOptimizerin the distributed TensorFlow 1.xmodel. 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_gradientsfunction to model.trainable_variables. Thus, after applying the function, the code calls the hvd.broadcast_variablesAPI to broadcast the variables and sets the boolean flag hvd_broadcast_doneto Trueto prevent repeated broadcasting.~~ variable broadcasting is explicitly executed at the first training epoch as shown in lines 31 to 34.

3. Overview

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 5 illustrates the overview of our automated code transformation approach for distributed training of DL models with Horovod. Our approach

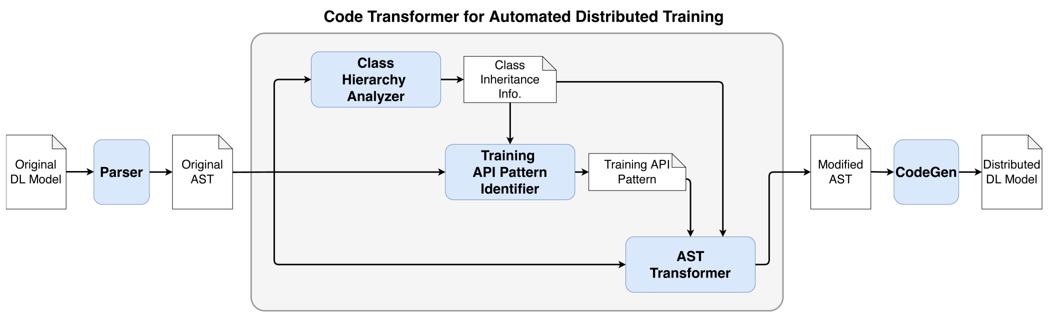


Figure 5: Overall structure of the automated transformation for distributed training

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

```
1 from tensorflow import keras
2
3 # `ResNet` inherits `keras.Model`
4 class ResNet(keras.Model):
5     def __init__(self, block_list):
6         ...
7
8 model = ResNet([2,2,2])
9 model.fit(x_train, y_train)
```

(a) Single-GPU DL model

```
1  from tensorflow import keras
2  import horovod.tensorflow.keras as hvd
3
4  class ResNet(keras.Model):
5      def __init__(self, block_list):
6          ...
7
8  model = ResNet([2,2,2])
9
10 model.fit(
11     x_train,
12     y_train,
13     callbacks=[hvd.callbacks.BroadcastGlobalVariablesCallback
    (0)])
```

(b) Distributed DL model

Figure 6: Code example of distributing a single-GPU DL model using a user-defined class

Figure 6a 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 6b demonstrates a modification of the model presented in Figure 6a 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 6a, 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 6a, 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

~~TensorFlow offers multiple APIs for defining the model structure and the training process. These APIs include low-level APIs such as tf.GradientTapeand high-level APIs such as tf.keras.Model. The choice of API depends on the model' complexity and the project's specific requirements~~

Our transformation approach needs to apply different transformation rules based on the API usage in the TensorFlow model. Figure 7 illustrates two TensorFlow model codes that use different APIs to define the training

```python
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

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

(b) Using high-level training API

Figure 7: TensorFlow model code example using two different API patterns

process. ~~Figure 7a explicitly repeats the training steps using the forloop and the GradientTapeinstance. On the other hand, Figure 7b uses the Keras library APIs, compileand 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~~ **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 7a. 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 8 illustrates a code example of the Session pattern. The ~~withstatement in line 1 creates an instance of~~ pattern creates the `Session` ~~class, and~~ object, which provides the `run` method ~~called in line 3 invokes the training computationon an isolated execution environment of the session instance. The runmethod is usually called multiple times through a loop statement to train models on several training batches.~~ to execute the model computation.

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

Figure 8: Session pattern code example

MonitoredSession Pattern~~.~~. ~~The MonitoredSession is another typical code pattern observed in TensorFlow 1.x models.~~ Figure 9 demonstrates a code example of the MonitoredSession pattern. This TensorFlow 1.x pattern bears similarities to the Session pattern, wherein the training computation is invoked directly. ~~However, instead of utilizing an instance of~~ Instead of using the `Session` ~~class~~object, the MonitoredSession pattern ~~leverages an instance of~~ codes use the `MonitoredSession` ~~class that provides hooks to perform some actions automatically on specific conditions during the training. The~~ object that provides hook methods. For instance, the code in Figure 9 uses the ~~`with`statement in line 3 creates an instanceof 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~~ to automatically save the model summaries after each training step. ~~TensorFlow provides the TensorBoardutility visualizing model summaries, so that model engineers can check training processes from the summaries.~~

```
summary_hook = SummarySaverHook(...)

with MonitoredTrainingSession(hooks=[summary_hook]) as
    mon_sess:
     while not mon_sess.should_stop():
         mon_sess.run(train_op, feed_dict=feed_dict)
```

Figure 9: 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 10 demonstrates an example of the GradientTape pattern. ~~Line 4 initiates~~ It creates the `GradientTape` ~~class instance~~ object through a `with` statement. ~~Once creating the~~ The ~~`GradientTape`~~`with` ~~instance, it watches all trainable variables by default and records operations executed on the variables within its context manager. The~~ statement body runs the model computation. Then, the ~~`gradient`~~`Optimizer` ~~method call in line 7 calculates and returns gradients from the recorded operations and the watched variables. Finally, in line 9, the~~ class instance calls the method `apply_gradients` ~~method of the Optimizerclass 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 11 represents an example of the Keras pattern. ~~Line 1~~ 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 10: GradientTape pattern code example

pattern defines the `ResNet` class inherited from the `keras.Model` ~~class. As described in Section [4], the~~ and uses it to construct the model object. The ~~keras.Modelclass provides a convenient way to construct and train models. Line 5 constructs a model as an instance of the ResNetclass. Finally, the~~ `fit` method ~~invoked in line 6 trains~~ is then invoked to automatically train the model with ~~the~~ a given dataset.

```
1 class ResNet(keras.Model):
2     def __init__(self, params):
3         ...
4 
5 model = ResNet([2, 2, 2], num_classes)
6 model.fit(dataset, epochs=50)
```

Figure 11: 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~~ based on a flat lattice domain composed

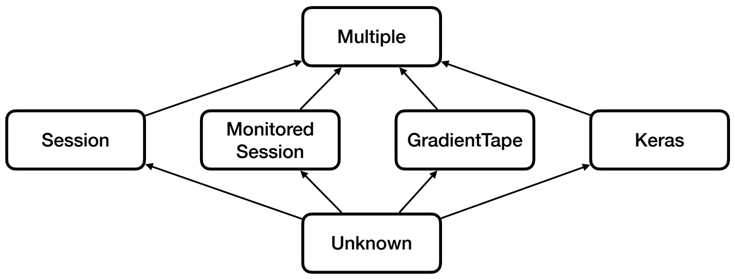


Figure 12: Flat lattice for the training API pattern identification

of the four training API patterns ~~, and~~ with two additional elements, *Unknown* and *Multiple* elements. The Unknown element represents that an AST does not match any of the four training API patterns, and the Multiple element represents that an AST matches multiple training API patterns. Figure 12 depicts the lattice structure. The lattice represents a partially ordered set ~~$(P, \sqsubseteq)$ of which~~ , where 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 ($\forall x, y \in P.\ x \sqsubseteq y \land x \neq y \rightarrow y \not\sqsubseteq x$), and transitive ($\forall x, y, z \in P.\ x \sqsubseteq y \land y \sqsubseteq z \rightarrow x \sqsubseteq z$).~~ We also define ~~an internal binary operation on the set P, denoted as the symbol~~ a join operation ($\sqcup$~~and named join, which computes the least upper bound *lub* of arbitrary two elements x and y in P as follows: $x \sqsubseteq lub \land y \sqsubseteq lub \land \forall ub \in \{e \mid x \sqsubseteq e \land y \sqsubseteq e\}.\ lub \sqsubseteq ub$. applies the join operation to produce one identification result from a model in which the statements match multiple training API patterns.~~) between any two elements on the lattice, which calculates the least element of their common successors. For example, the Session element results from joining Session and Unknown elements, and the Multiple element results from joining MonitoredSession and Keras elements. By utilizing the join operation, Training API Pattern Analyzer ensures that it always identifies a single API pattern even when the statements of a model match multiple training API patterns.

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

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
```

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 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

```
trans_S(`import {aliases}`, ctx):
  let ctx_1 = trans_A(`{aliases}`, ctx).context
  if diff(ctx_1, ctx) == ["tensorflow" -> `{id}`]:
    return (`import {aliases}; import horovod.tensorflow as hvd`, ctx_1)
  else:
    return (`import {aliases}`, ctx_1)
```

(a) Transform function ~~description~~of import statements

```
import tensorflow as tf
```

(b) Original DL python code

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

(c) Transformed DL python code

Figure 13: Transform function example

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 functions take a Python AST and a ~~*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~~ object as inputs, and return a Python AST and a context object as output. A context object is a mapping from strings to Python identifier ASTs; context objects store and propagate necessary identifiers.

This enables (pure) transform functions to ~~utilize contextual information outside their input ASTs and~~ pass relevant contextual information to subsequent transform ~~functions.~~ function calls and utilize the information from outside their input AST.

~~We define the transform functions as a collection of partial transform functions, of which one example is illustrated in Figure 13. These partial functions operate only on ASTs that match a specified code pattern in the input ASTparameter position. Any input ASTs that do not match the pattern remain unmodified. For example , the function in Figure 13 matches only~~ Figure 13a illustrate the pseudocode of an example transform function that transforms an import statement. The transform function ~~`import`~~`trans_S` ~~statements and transforms them accordingly. The function takes the transform~~ gets an input AST and a 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~~ object. In the pseudocode, we use back-quoted notation ~~`LET-IN`~~`` `import {aliases}` `` ~~and IF-THEN-ELSEto specify different outputs depending on certain conditions. In Figure 13, the function body uses a LET-INexpression to store the updated transform context by another transform function applied to the sub phrase *alias**~~ to ~~σ_I. Also, it uses an~~ represent a Python AST object; the brace-surrounded expression represents a child AST. Line 2 creates a new context object by calling the transformation function on the child AST, ~~`IF-THEN-ELSE`~~`` `{aliases}` `` ~~expression to return different results depending on whether the~~ . Then, line 3 computes the difference between the original context object ~~`importstatement imports the`~~ `ctx` and the new context object ~~`tensorflowmodule or not.`~~

~~`Original DL python code Transformed DL python code Example code transformation result`~~

~~`Figure ?? shows the example code transformation result by the partial transform function in Figure 13. Because the statement in Figure 13b 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` $\sigma_I$ `that includes all the modules and their alias identifiers imported by the statement. Next, the function calculates the difference between` $\sigma_I$ `and the input transform context` $\sigma$`. Suppose the difference contains the new entry for the string`~~ `ctx_1`. If the new context additionally stores the "~~`tensor_flow`~~`tensorflow`" ~~. In that~~ entry, it infers that the import statement imports the TensorFlow library. In this case, the transform function ~~returns a statement list containing the original statement and a new statement~~ places a new import statement, `import horovod.tensorflow as hvd`~~to add the import statement for the horovodmodule after the original statement, as shown in Figure 13c~~, right after the TensorFlow import statement and returns the modified AST. Otherwise, the function ~~returns a statement list containing only the original statement remaining~~ leaves the input AST unchanged.

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

6.2. Transformation Rules for API Patterns

6.2.1. Rules for the Session Pattern

~~Figure~~

Figure 14 illustrates the ~~required transformation of~~ transform function for the Session pattern ~~using two code examples, 14b the original training code and 14c its distributed training version. One of the primary transformations involves adjusting the learning rate argument in the~~ with code examples. The transform function described in Figure 14a transforms the original code in Figure 14b to the distributed code in Figure 14c. In the `Optimizer` ~~constructor call, which is achieved by multiplying it by the number of GPUs. The learning rate can be passed as the first positional argument in the constructor call or as the keyword argument~~ object's constructor, the `learning_rate` ~~. The other transformation uses the Horovod-provided~~ argument is multiplied by ~~`DistributedOptimizer`~~`hvd.size()` ~~instance instead of the original~~. Then, the constructed `Optimizer` ~~instance.To create the~~ object is wrapped with the Horovod API, ~~`DisributedOptimizer`~~`DistributedOptimizer`~~instance, the original Optimizeris passed as an argument to its constructor.~~ .

~~= tensorflow.keras.optimizers.Optimizer learning_rate 1 ≤ i ≤ k (, = hvd.DistributedOptimizer(),)~~

~~(, = hvd. DistributedOptimizer(),)~~

~~Session pattern transform function: Optimizer learning rate scaling and wrapping~~

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

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

```
trans_S(`{id_r} = {expr}({a_1}, ..., {a_n})`, ctx):
  if isSubclass(`{expr}`, `keras.optimizer.Optimizer):
    for i in 1...n:
      if `{a_i}` == `learning_rate = {lr}`:
        let new_stmt =
          `{id_r} = {expr}({a_1}, ..., {a_i}*hvd.size(), ..., {a_n})`
        let wrap_stmt =
          `{id_r} = hvd.DistributedOptimizer({id_r})`
        return (new_stmt; wrap_stmt), ctx["optimizer"->`{id_r}`]
```

(a) Transform function for Session pattern

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

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

(b) 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)
```

(c) Distributed Session pattern example

Figure 14: Example transformation of the Session pattern

~~that stores the result of a function call expression into a variable. Initially, the rule examines~~ Given a call assignment statement, the transform function checks whether the callee function is the constructor of ~~either~~ the `Optimizer`~~class or any of its subclasses. The predicate 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 learningrate and the return value of~~ 's subclasses. If true, the ~~`horovod`~~`for` loop searches for the `learning_rate` argument from the function arguments to construct a new call assignment statement with the `learning_rate` argument multiplied by hvd.size(). ~~Otherwise, the rule adjusts the first argument instead. Following the adjustment of the learning rate, the rule adds a new statement : id_r =~~The function finally returns the new statement concatenated by another assignment statement that wraps the ~~hvd.DistributedOptimizer(id_r)~~Optimizer object with DistributedOptimizer. ~~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 15 describes an example transformation of the MonitoredSession pattern code. The MonitoredSessionconstructor optionally requires a list of SessionRunHookobjects as a keyword argument hooks. To ensure consistent global variable initialization of all processes in the distributed training, the list needs to contain a BroadcastGlobalVariableHookobject for the global variable broadcasting, as shown in Figure 15c. The instance can be created by calling the constructor of the BroadcastGlobalVariableHookclass 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 ~~' , ' ,~~
~~(' ' ,)~~
~~=~~

15 illustrates the transform function for the MonitoredSession pattern with code examples. In the example code, a ~~tensorflow.train.MonitoredSession~~with statement first constructs the ~~hooks~~MonitoredTrainingSession ~~$1 \leq i \leq k$ (,~~object and binds it to the name mon_sess. We refer to the pair of the object and its name appearing in the with statement as a *with item*. The MonitoredTrainingSession object constructor is transformed to append hvd.BroadcastGlobalVariablesHook(0)~~MonitoredSession pattern transform function: Modifying StopAtStepHookinstance~~ to the end of the hooks argument.

~~Figure?? formalizes the transform functionsfor the MonitoredSession pattern~~Figure 15a describes two transform functions. The first ~~transform functionis responsible for matching~~ function, ~~with~~trans_S~~statements and transforming a list of the~~, takes a with~~_item , each of which represents a variable and an assigned expression. The transform function denoted as in line 2 applies the second transform function to each~~ statement AST as an input, then applies ~~with_item~~trans_W~~. As a result, the first transform function returns a new~~ to its with items. The second function, ~~with~~trans_W~~statement containing a list of the transformed~~,

```
trans_S(`with {with_items} : {stmts}`, ctx):
  let `{with_items_1}`, ctx_1 = trans_W(`{with_items}`, ctx)
  let `{stmts_1}`, ctx_2 = trans_S(`{stmts}`, ctx_1)
  return (`with {with_items_1} : {stmts_1}`, ctx_2)
trans_W(`{e_1}({a_1, ..., a_n}) as {id}`, ctx):
  if isSubclass(e_1, tf.train.MonitoredSession):
    for i in 1...n:
    if `{a_i}` == `hooks = {e_i}`:
      let new_hooks =
        `{e_i}.append(hvd.BroadcastGlobalVariableHook(0))`
      let new_with =
        `{e_1}({a_1, ..., hooks={new_hooks}, ..., {a_n})`
      return (new_with, ctx["monsess"->`{id}`])
```

(a) Transform function for MonitoredSession pattern

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

(b) 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()
```

(c) Distributed MonitoredSession pattern example

Figure 15: Example transformation of the MonitoredSession pattern

first checks if the given with item constructs the ~~`with_item`~~`tf.train.MonitoredSession` ~~. The second transform function transforms the~~ subclass object. If true, the function finds the ~~with _item~~ `hooks` ~~. Among many forms of the~~ argument from the constructor call and appends the ~~`with_item`~~`hvd.BroadcastGlobalVariablesHook(0)`~~, the functionmatches those that assign a function call resultto a variable. The second line checks whether the callee function of the~~. The function returns the modified with item AST to the ~~`with_item`~~`trans_S` ~~is a constructor of the subclass of the~~ function. As a result, the ~~`MonitoredSession`~~`trans_S` ~~class. If then, the transformation appends a~~ function returns the newly modified ~~`BroadcastGlobalVariablesHook`~~`with` ~~object to the keyword argument~~ statement with its body statements recursively transformed by ~~hooks.~~ trans_S.

6.2.3. Rules for the GradientTape pattern

```
1 import tensorflow as tf
2
3 with tf.GradientTape() as tape:
4   probs = model(images)
5   loss_value = loss(labels, probs)
6
7 grads = tape.gradient(loss_value, model.trainable_variables)
8 opt.apply_gradients(zip(grads, model.trainable_variables))
```

(a) GradientTape pattern example

```
1  import tensorflow as tf
2  import horovod.tensorflow as hvd
3  hvd_broadcast_done = False
4
5  with tf.GradientTape() as tape:
6    probs = model(images)
7    loss_value = loss(labels, probs)
8  tape = hvd.DistributedGradientTape(tape)
9  grads = tape.gradient(loss_value, model.trainable_variables)
10 id_new = zip(grads, model.trainable_variables)
11 opt.apply_gradients(id_new)
12
13 global hvd_broadcast_done
14 if not hvd_broadcast_done:
15   hvd.broadcast_variables([x[1] for x in id_new], root_rank
       =0,)
16   hvd.broadcast_variables(opt.variables(), root_rank=0,)
17   hvd_broadcast_done = True
```

(b) Distributed GradientTape pattern example

Figure 16: Example transformation of the GradientTape pattern

Figure 16 illustrates an example GradientTape pattern code and its distributed version. ~~For the distributed training, the model needs to utilize a~~ There are two main transformations applied in the codes. First, the ~~DistributedGradientTapeinstance in training instead of the~~ GradientTape ~~instance, as shown in lines 8 and 9 in Figure 16b. 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 16b represent the code that broadcasts the variables depending on the value of~~ object defined by the ~~hvd_broadcast_done~~with ~~. Once finishing the broadcasting, the~~ statement is wrapped with ~~hvd_broadcast_doneis set to Falseto prevent further broadcasting.~~

~~= ′, ′, ↦ (′ ′,~~ hvd.DistributedGradientTape~~()~~ ~~,) (′ ′,) GradientTape pattern transform function: utilizing the DistributedGradientTapeinstance~~

~~Figure ?? depicts the transformation for the GradientTape pattern utilizing the~~ API. The transformation also re-assigns the ~~DistributedGradientTape~~grads ~~instance. The transform function matches~~ variable with the result of calling the ~~with statements and updates the transform context by transforming the list of with_item~~gradient ~~. The updated transform context contains a mapping from the string graident_tapeto a variable if~~ method before using it in the ~~with_item~~apply_gradients ~~creates a GradientTapeinstance and assigns it to the variable. Then, the function injects a new assignment statement, *id* = hvd.DistributedGradientTape(id), right~~ method. Second, the code for broadcasting variables is appended after the `with` statement, ~~which replaces the instance of the variable with a newly created DistributedGradientTapeinstance for further uses in subsequent statements~~as shown in lines 13 to 17 in Figure 16b.

~~= ()~~

```
trans_S(`with {with_items} : {stmts}`, ctx):
  let `{with_items_1}`, ctx_1 = trans_W(`{with_items}`, ctx)
  let `{stmts_1}`, ctx_2 = trans_S(`{stmts}`, ctx_1)
  if diff(ctx_1, ctx) == ["gradient_tape" -> `{id}`]:
    let wrap_stmt =`{id} = hvd.DistributedGradientTape({id})`
    return (`with {with_items_1} : {stmts_1}; {wrap_stmt}`, ctx_2)
  else:
    return (`with {with_items_1} : {stmts_1}`, ctx2)
```

Figure 17: First transform function for GradientTape pattern

Figure 17 describes the transform function responsible for the first transformation. After transforming the with items and body statements, the function checks if the with items define a new ~~. apply_gradients~~gradient_tape ~~(, ,~~item. If true, the function constructs a statement that wraps the ~~global hvd_braodcast_done~~gradient_tape ~~,~~item with the ~~if not hvd_broadcast_done:~~hvd.~~broadcas x in , root_rank=0)~~DistributedGradientTape ~~,~~ API. Finally, the function returns the new ~~hvd. broadcast_variables(.variables(), root_rank=0)~~with

~~, hvd_broadcast_done = True,)~~ statement concatenated with the new wrapping statement.

```
trans_S(`{id} = {expr}({a_1}, ..., {a_n})`, ctx):
if ctx["optimizer"] == `{id_t}` && `{expr}` == `{id_t}.apply_gradients`:
  let id_z = `id_new`
  let input_stmt = `{id} = {expr}({a_1}, ..., {a_n})`
  let broadcast_stmts =
    `{id} = {expr}({a_1}, ..., {a_n})`;
     global hvd_broadcast_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`
  return (input_stmt; broadcast_stmts, ctx)
```

Figure 18: ~~GradientTape pattern~~ Second transform function ~~: broadcasting trainable global variables~~for GradientTape pattern

Figure~~?? formalizes the partial transform function that appends the variable broadcasting code after the apply_gradientsmethod call~~ [18] describes the transform function responsible for the second transformation, which transforms a call assignment statement. The function ~~matches assignment statements that assign a function call result. When a variable id_t stores the~~ first searches for an ~~`Optimizer`~~`optimizer` ~~instance and the callee function is~~ variable in the context object. Then, it checks if the input statement calls the ~~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~~ method of the ~~`global`~~`optimizer` ~~statement to refer to the global broadcast flag variablehvd_broadcast_doneand 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~~variable. If true, the function places the code for broadcasting variables right after the input statement.

6.2.4. Rules for the Keras pattern

```
trans_S(`{expr}({a_1}, ..., {a_n})`, ctx):
  let `{id_m}` = ctx["model"]
  if `{expr}` == `{id_m}.fit`:
    let cb_stmt =
      `cb = [hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0)]`
    let new_stmt =
      `{expr}({a_1}, ..., {a_n}, callbacks=cb)`
    return (cb_stmt; new_stmt, ctx)
```

(a) Transform function for Keras pattern

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

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

model.fit(x_train, y_train)
```

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

(c) Distributed Keras pattern example

Figure 19: Example transformation of the Keras Pattern

~~Figure 19 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 fitmethod. To distribute the model's training with Horovod, the fitmethod needs to take a callback BroadcastGlobalVariablesCallbackas the keyword argument callbacks, shown~~ Figure 19 illustrates the transform function for the Keras pattern with code examples. The function transforms the original code in Figure 19b to the distributed code in Figure 19c. ~~When training starts, the callback ensures consistent initialization of all processes by broadcasting initial global variable states from a source to the other processes.~~

~~= (~~The transformation first constructs a new list that contains a callback object, ~~) .fit callbacks 2 ≤ i ≤ k~~ `BroadcastGlobalVariablesCa` ~~= hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=~~`0)`~~,~~ . Then, the callback list is propagated to the ~~if hvd. rank() == 0: cb.append(), (··· ~~`fit`~~ ···~~ function as a new ~~cb~~`callback` ~~··· ,) (cb = hvd. callbacks.BroadcastGlobalVariables ··· callbacks cb),) Keras pattern transform function~~ argument.

~~Figure ?? formalizes the transform function for the Keras pattern code.~~ The transform function ~~matches function call statements for the~~ takes a function call statement as an input. The context object contains information about the variable that holds the ~~fitmethod of model objects by checking whether the receiver object of the fitmethod call is an instance of a subclass of keras.~~ `Model` ~~. Then, if the function call already takes the callbackskeyword argument, the transform function changes the~~ subclass object. Using the information, the function ~~call to three statements. The first and second statements create a temporary variable~~ first checks if the callee function is the ~~cb~~`fit` ~~for the later use of the function call in the third statement . Note that the second statement appends the original callbacks to the~~ method of a model object. If true, the function constructs a statement that creates a list of a ~~cb~~`BroadcastGlobalVariablesCallback` ~~variable only when the hvd. rank()is zero, which enables the original callbacks to be called only on the one process. If~~ object and assigns it to a new variable. Then, it modifies the input call statement to propagate the list as the keyword argument~~does not exist in the function call, the transform function propagates the callback as the~~ , `callbacks`~~keyword argument, shown in Figure 19c~~ . Finally, the function replaces the input call statement with the sequence of two statements.

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~~Efficiency) How much more efficiently do automatically transformed distributed training codes perform compared to original single-GPU-based training codes?

To answer the questions, we implemented an automatic code transformation tool and applied the tool to 16 TensorFlow DL models. We ~~collected~~ colleced 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].

We do not include large-scale real-world models in our evaluation because they depend on external training libraries or non-Python scripts in the training process. Since our work focuses on the transformation of pure TensorFlow DL models, applying our tool to other types of models lies beyond this paper's scope. We discuss this limitation and a possible solution in Section 9.

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	○

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

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~~ For the other models, ~~we referred~~ the first author manually transforms the code into distributed training code by referring to the Horovod documentation~~to manually transform them into distributed training versions and used those as the correct results~~ [22]. Precisely, the first author followed step-by-step instructions in the Horovod documentation to convert each evaluation target code into its distributed version. Subsequently, the second author independently compared them with the codes transformed by our tool. If the two codes are identical except for minor syntactic differences such as whitespaces or variable declaration order, we concluded that our tool had correctly transformed the original model code.

The results in 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~~ show that our tool correctly ~~transforms the models. As shown in the table, our tool correctly transformed 15 out of the 16 models and~~ transformed **15 out of 16** target models. The tool failed to transform only the SimpleCNN-GradientTape-1 model. In the failed case, our tool raised a transformation failure error because 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 20 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 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~~ The case is rare since the other 11 GradientTape models use the `Model` ~~instance~~API to construct models.

```python
# Model object is not used, instead a function used
def conv_net(x):
    x = tf.reshape(x, [-1, 28, 28, 1])
    conv1 = conv2d(x, weights['wc1'], biases['bc1'])
    conv1 = maxpool2d(conv1, k=2)
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
    conv2 = maxpool2d(conv2, k=2)
    fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().
   as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'
   ])
    fc1 = tf.nn.relu(fc1)
    out = tf.add(tf.matmul(fc1, weights['out']), biases['out'
   ])
    return tf.nn.softmax(out)

optimizer = tf.optimizers.Adam(learning_rate)

def run_optimization(x, y):
    with tf.GradientTape() as g:
        pred = conv_net(x)
        loss = cross_entropy(pred, y)
    trainable_variables = list(weights.values()) + list(
   biases.values())
    gradients = g.gradient(loss, trainable_variables)
    optimizer.apply_gradients(zip(gradients,
   trainable_variables))
    # cannot perform variable broadcast with Model.variables

# training loop
for step, (batch_x, batch_y) in enumerate(train_data.take(
   training_steps), 1):
    run_optimization(batch_x, batch_y)
```

Figure 20: Training code of SimpleCNN-GradientTape-1 model

In conclusion, the results of the RQ1 experiment demonstrate that our code transformation approach successfully transforms most TensorFlow DL models. However, our tool fails in one case where the code utilizes the training APIs in an uncommon manner, which is a limitation of our approach.

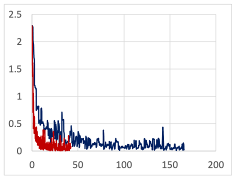
Table 3: Training time comparison results

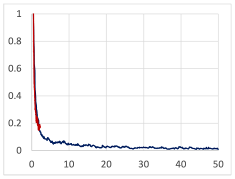
Model Name	Non-distributed training time (s)	Distributed training time (s)	Speedup
LSTM-MNIST	78.675	11.951	×6.58
SimpleCNN-GradientTape-2	3.192	1.753	×1.82
VGG-CIFAR10	967.076	299.229	×3.23
Play-with-MNIST	148.101	80.040	×1.85
Linear-Regression	0.607	0.371	×1.63
Fashion-MNIST	110.274	29.294	×3.76
CIFAR10-VGG16	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

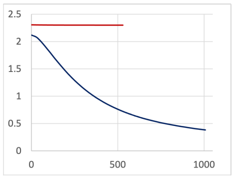
7.2. RQ2: ~~Effectiveness~~ Efficiency of the automatically distributed training

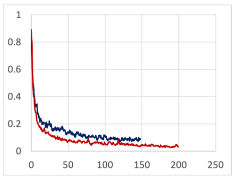
For the second research question, we conducted a comparative analysis between the training time of the automatically transformed models and their respective original models. We calculate the training time as the time taken to reach minimum losses in training. To measure the losses, we manually injected the TensorBoard [23] 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.

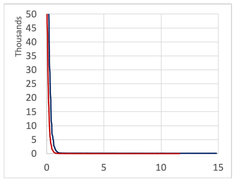
~~We summarize the training time comparison results in Table 3. The first column presents the model name, the second and third columns present the training time of the model, and the last column presents the training speed-up of the distributed model. The comparison results indicate that the~~ The result of the RQ2 experiment shows that the distributed training of the ~~distributed models~~ transformed model is, on average, ~~approximately 2.28 times faster~~ **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~~. When the distributed training models outperform the original ones, the **performance enhancement ranges from x1.37 to x6.58**. We omit the distributed

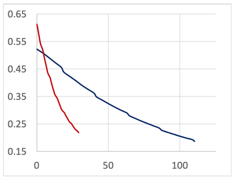

(a) LSTM-MNIST

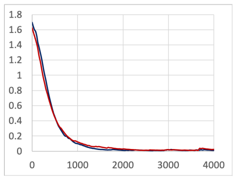

(b) SimpleCNN-GradientTape-2

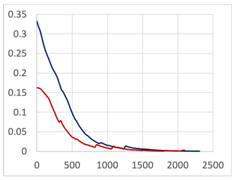

(c) VGG-CIFAR10

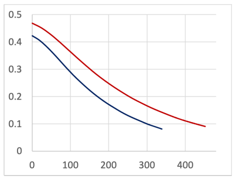

(d) Play-with-MNIST

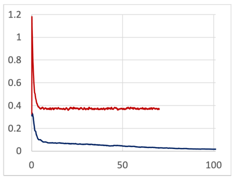

(e) Linear-Regression

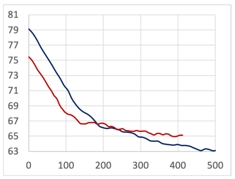

(f) Fashion-MNIST

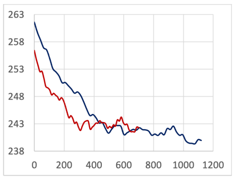

(g) CIFAR10-VGG16


(h) Inception-network


(i) RNN-Sentiment-Analysis


(j) Stacked-LSTM-ColorBot


(k) Auto-Encoder


(l) Variational-Auto-Encoder

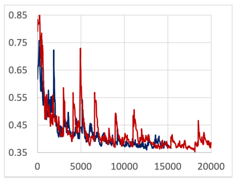

(m) DCGAN

Figure 21: 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)

training time of the Stacked-LSTM-ColorBot ~~model~~ case from the result because the loss of ~~the first distributed training epoch of the model is minimal~~ ~~.~~

~~Interestingly, distributed training~~ its first epoch already reached minimal in the experiment. We conclude that while the distributed training of automatically transformed models 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~~faster training time in every case, it enhances the training performance by x2.28 on average.

Our further investigation shows that the distributed training of transformed models does not present the same accuracy as the original ones. Figure 21 illustrates the ~~changes~~ change 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~~ over time during training. 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~~ In six out of 13 models, distributed training reaches a higher loss compared to the original model. For instance, in the ~~case of the~~ original VGG-CIFAR10 model, ~~we observed that~~ the loss gradually decreases during ~~the trainingprocess without distributed training~~training. However, ~~when adopting~~ in its distributed training, the loss ~~did not decrease significantly and remained almost~~ does not meaningfully decrease and remains unchanged, with only minor fluctuations. In conclusion, while our approach may improve the training performance by automatically distributing them, it does not always ensure that the resulting model has the same accuracy to its original one.

~~Distributed training often requires additional tuning of hyperparameters to achieve positive effects~~

Tuning training hyperparameters can improve the training performance of automatically distributed models. We conducted ~~distributed training~~ an additional experiment on the VGG-CIFAR10 model by applying three different learning rate parameters and comparing their training performances. Figure 22 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.

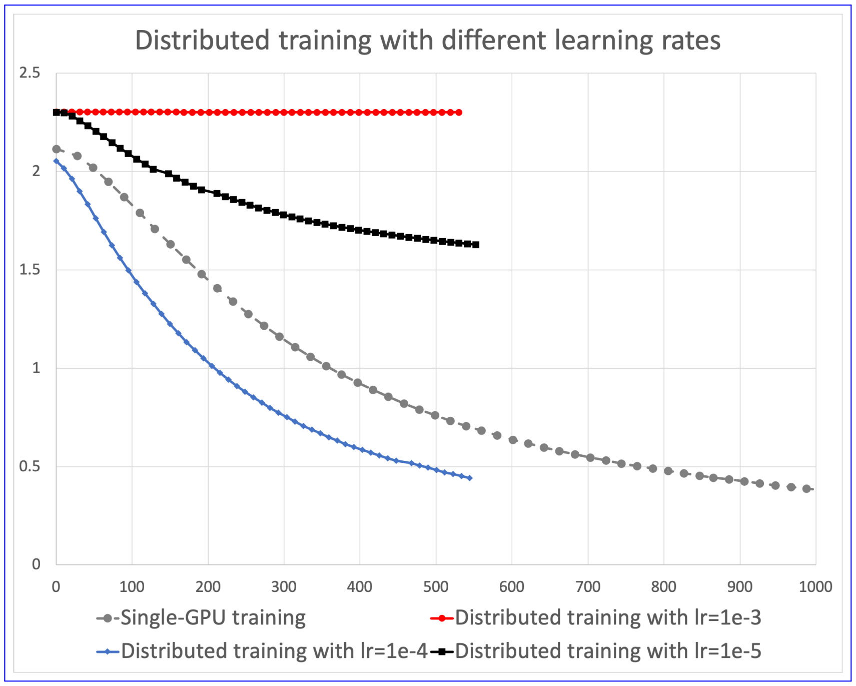


Figure 22: Distributed training on the VGG-CIFAR10 model with different learning rates

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~~ developers may need to ~~tune~~ configure hyperparameters differently from ~~non-distributed~~ original single-GPU-based models to obtain the benefits of distributed training.

~~Distributed training on the VGG-CIFAR10 model with different learning rates~~ There are several previous works [24][25][26] on automatically tuning the hyperparameters of specific settings. However, there is yet to be a known automated method that finds the optimal hyperparameters of distributed models. Engineers may still need to adjust hyperparameters by trial and error to optimize the training performance of distributed models generated by our approach. Future works can benefit engineers by enabling automatic hyperparameter tuning for distributed models.

8. Related Work

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 [27] 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 [28]. 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. Automatically Distributing Deep Learning Models

Several works proposed techniques for automatically optimizing the distributed training of DL models. A line of work searches for optimal parallelization plans before the runtime of DL model training codes. Megatron-LM [29] is one of the earliest works on automatically finding efficient DNN model training parallelization plans. The Megatron-LM framework distributes operations inside the layers of transformer networks by adding synchronization primitives. Pipedream [30] is an asynchronous distributed training framework that improves parallel training throughput by adding pipelining and intra-batch parallelism. PipeDream also minimizes communication costs by partitioning the training computation based on a profiling run on a single GPU. Alpa [31] is an automatic model-parallel training framework that organizes different parallelization techniques into a hierarchy and maps them to the hierarchical structure of computing devices. This allows Alpa to optimize plans for both inter-operator parallelization and intra-operator parallelization. Lin et al. [32] introduce new primitive operators that allow domain experts to compose their own search space, and their proposed framework generates an efficient parallelization plan for the DNN model training. AutoDDL [33] adds new tensor operators to expand the search space for parallelization strategies. The framework uses an analytical performance model with a Coordinate Descent-based search algorithm to optimize the communication cost.

Another line of work focuses on optimizing resource usage during the training runtime. DAPPLE [34] is a synchronous distributed training framework that combines data and pipeline parallelism for large DNN models, which finds the optimal parallelization plan with a novel parallelization strategy planner and a new runtime scheduling algorithm to reduce memory usage. Tiresias [35] is a GPU cluster manager that efficiently schedules DL training jobs to reduce their job completion times. Because predicting DL jobs' completion time is challenging, the authors propose two scheduling algorithms tailored to minimize the average job completion times.

Compared to these works, our work's primary technique is to boost the DL model training by automatically transforming single-GPU-based training codes into distributed ones. The aforementioned works require developers to write distributed training codes according to the proposed framework manually. Our work utilizes the Horovod library's data-parallel distributed training to distribute existing single-GPU-based DL training codes without rewriting them. While previous works' primary technique is to distribute low-level, primitive computing operations over multiple GPUs, our primary technique is to transform a single-GPU-based DL training code to a distributed version at the Python source code level. By reusing the Horovod library and converting the training code at the source code level, we allow non-expert developers to distribute the training code efficiently without manually rewriting the original training code.

8.3. Code Transformation

Code transformation is techniques that modify code into a different form. Visser devised a taxonomy [36] 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 [37] devised a framework that optimizes PySke programs by automatically rewriting terms in the programs based on transformation rules they define. Haryono et al. [38] developed MLCatchUp, a code transformation tool that enables Python machine learning programs to migrate from deprecated APIs to new and stable APIs automatically. ~~Reed~~ In Zhang et al. ~~[39] developed torch.fx, a Python library for capturing and transforming PyTorch programs~~[40], the authors develop AST-rewriting operations to detect non-idiomatic code for each Pythonic idiom and refactor non-idiomatic Python codes into idiomatic codes. Rózsa et al. [41] introduce a framework that transforms conditional branches into structural pattern-matching without changing the original behavior. This improves the readability and maintainability of codes written in legacy Python. Tangent [42] is a new library that transforms a subset of Python and NumPy code to perform automatic differentiation on a source code level. Compared to these works, our work targets TensorFlow DL models written in Python, and provides concrete and correct transformation rules for their distributed training. To our knowledge, our work is the first to utilize the Python code transformation technique to distribute single-GPU-based DL training codes automatically.

9. Conclusion

~~We propose the~~ This paper proposes an automated approach to transform TensorFlow DL models written in Python to models training on multiple GPUs. We ~~defined four common training patterns for~~ categorized TensorFlow DL models ~~and formal code~~by four patterns of training API usage and devised a static analysis technique that identifies the training pattern of the given model code. Then, we defined code transformation rules for each ~~pattern to parallelize training via Horovod APIs. Also, we developed~~ training API pattern, which parallelize the training process via Horovod library APIs. To this end, we implemented a code transformation tool that takes a ~~TensorFlow DL model~~ single-GPU-based DL model training code, 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~~ , and applies the corresponding source code transformation to output a distributed training code for the model.

We conducted two experiments to evaluate our proposed approach. First, we show that our approach ~~is practical in correctly transforming~~ correctly transforms 15 out of 16 open-source TensorFlow ~~DL models. We also showed~~ LD models. Second, we show that our approach ~~is effective in~~ efficiently distributes the training process so that the transformed models train about 2.28 times faster than the original models. ~~We believe that our tool reduces model engineers' burdens~~ By an additional experiment, we claim that the hyperparameters of transformed models can be tuned to obtain better training speeds. As a result, we argue that our approach reduces developers' burden in rewriting models ~~in accordance with the documentation of distributed training libraries to parallelize training .~~ for distributed training.

We still leave limitations for further studies. Since many state-of-the-art models and real-world DL applications depend on external training libraries or non-Python scripts to train the model, we need to extend our tool to identify API patterns of various training libraries and transform them into distributed versions. We also expect that our code transformation approach can be extended to support other deep learning libraries, such as PyTorch [3] , and to other distributed training frameworks, such as DeepSpeed [15]. Finally, to correctly transform DL training codes embedded in real-world applications, future work can develop a static analysis technique that identifies only related parts to DL model training from complex multi-lingual applications.

The training performance of automatically transformed models could be further improved by automatically finding optimal hyperparameters. Previous works [24][25][26] have proposed techniques automated to tune hyperparameters of specific models. Incorporating the techniques into the pipeline of our approach, we can further improve the distributed training performance in a fully automated way. By addressing these future directions, we believe that the DL developer community can benefit from a wide range of advantages our code transformation approach provides.

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] T. H. Authors, Horovod with tensorflow (2019).
URL https://horovod.readthedocs.io/en/stable/tensorflow.html

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

[24] D. Yogatama, G. Mann, Efficient Transfer Learning Method for Automatic Hyperparameter Tuning, in: S. Kaski, J. Corander (Eds.), Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, Vol. 33 of Proceedings of Machine Learning Research, PMLR, Reykjavik, Iceland, 2014, pp. 1077–1085.
URL https://proceedings.mlr.press/v33/yogatama14.html

[25] P. Neary, Automatic hyperparameter tuning in deep convolutional neural networks using asynchronous reinforcement learning, in: 2018 IEEE International Conference on Cognitive Computing (ICCC), 2018, pp. 73–77. doi:10.1109/ICCC.2018.00017.

[26] P. Koch, O. Golovidov, S. Gardner, B. Wujek, J. Griffin, Y. Xu, Autotune: A derivative-free optimization framework for hyperparameter tuning, in: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 443–452. doi:10.1145/3219819.3219837.
URL https://doi.org/10.1145/3219819.3219837

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

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

[29] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, B. Catanzaro, Megatron-lm: Training multi-billion parameter language models using model parallelism (2020). arXiv:1909.08053.
URL https://arxiv.org/abs/1909.08053

[30] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, M. Zaharia, Pipedream: generalized pipeline parallelism for dnn training, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 1–15. doi:10.1145/3341301.3359646.
URL https://doi.org/10.1145/3341301.3359646

[31] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, I. Stoica, Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning, in: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), USENIX Association, Carlsbad, CA, 2022, pp. 559–578.
URL https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

[32] Z. Lin, Y. Miao, Q. Zhang, F. Yang, Y. Zhu, C. Li, S. Maleki, X. Cao, N. Shang, Y. Yang, W. Xu, M. Yang, L. Zhang, L. Zhou, nnScaler: Constraint-Guided parallelization plan generation for deep learning training, in: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), USENIX Association, Santa Clara, CA, 2024, pp. 347–363.
URL https://www.usenix.org/conference/osdi24/presentation/lin-zhiqi

[33] J. Chen, S. Li, R. Guo, J. Yuan, T. Hoefler, Autoddl: Automatic distributed deep learning with near-optimal bandwidth cost, IEEE Transactions on Parallel and Distributed Systems PP (2024) 1–14. doi:10.1109/TPDS.2024.3397800.

[34] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, L. Diao, X. Liu, W. Lin, Dapple: a pipelined data parallel approach for training large models, in: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 431–445. doi:10.1145/3437801.3441593.
URL https://doi.org/10.1145/3437801.3441593

[35] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, C. Guo, Tiresias: A GPU cluster manager for distributed deep learning, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), USENIX Association, Boston, MA, 2019, pp. 485–500.
URL `https://www.usenix.org/conference/nsdi19/presentation/gu`

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

[37] 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.

[38] 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.

[39] 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.

[40] Z. Zhang, Z. Xing, X. Xia, X. Xu, L. Zhu, Making python code idiomatic by automatic refactoring non-idiomatic python code with pythonic idioms, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Association for Computing Machinery, New York, NY, USA, 2022, p. 696–708. `doi:10.1145/3540250.3549143`.
URL `https://doi.org/10.1145/3540250.3549143`

[41] B. Rózsa, G. Antal, R. Ferenc, Don't diy: Automatically transform legacy python code to support structural pattern matching, in: 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), 2022, pp. 164–169. `doi:10.1109/SCAM55253.2022.00024`.

[42] B. van Merrienboer, D. Moldovan, A. Wiltschko, Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming, in: S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (Eds.), Advances in Neural Information Processing Systems, Vol. 31, Curran Associates, Inc., 2018.
URL `https://proceedings.neurips.cc/paper_files/paper/2018/file/748d6b6ed8e13f857ceaa6cfbdca14b8-Paper.pdf`