Automated Code Transformation for Distributed Training of TensorFlow DL Models

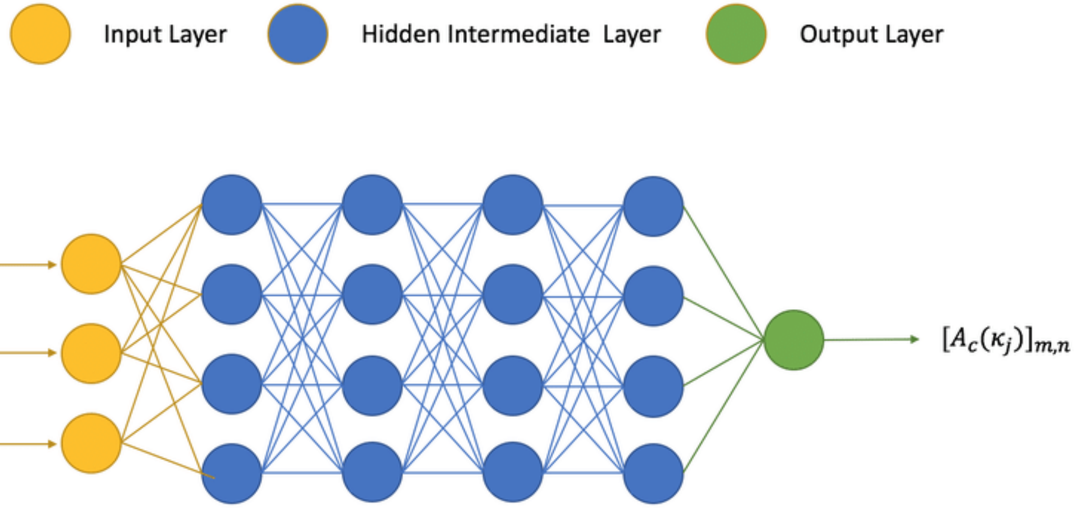
Yusung Sim[1], Wonho Shin[1], Shin Young Ahn[2], Sungho Lee[3], Sukyoung Ryu[1]

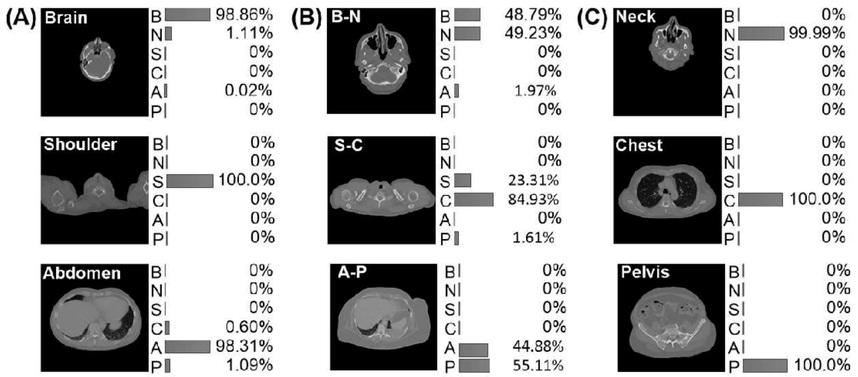
[1]School of Computing, KAIST

[2]Artificial Intelligence Research Laboratory / Supercomputing Technology Research Center, ETRI

[3]Department of Computer Science and Engineering, Chungnam National University

Deep Learning(DL)





Prediction of Discretization of GMsFEM Using Deep Learning - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/An-illustration-of-a-deep-neural-network_fig4_333049256 [accessed 11 Jul, 2022]
Cho, J., Lee, K., Shin, E., Choy, G., & Do, S. (2015). Medical Image Deep Learning with Hospital PACS Dataset. *ArXiv, abs/1511.06348*.

DL Model Development in Practice

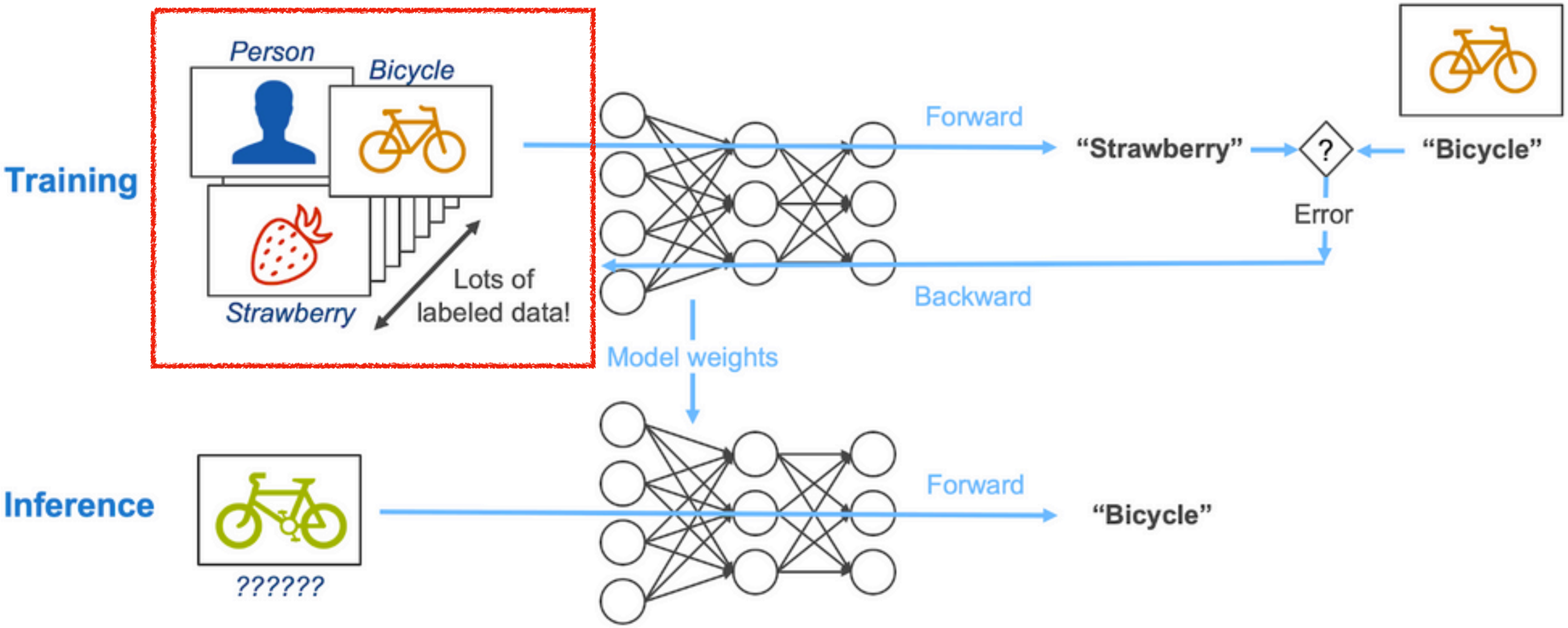

```python
import tensorflow as tf

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])

model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
```

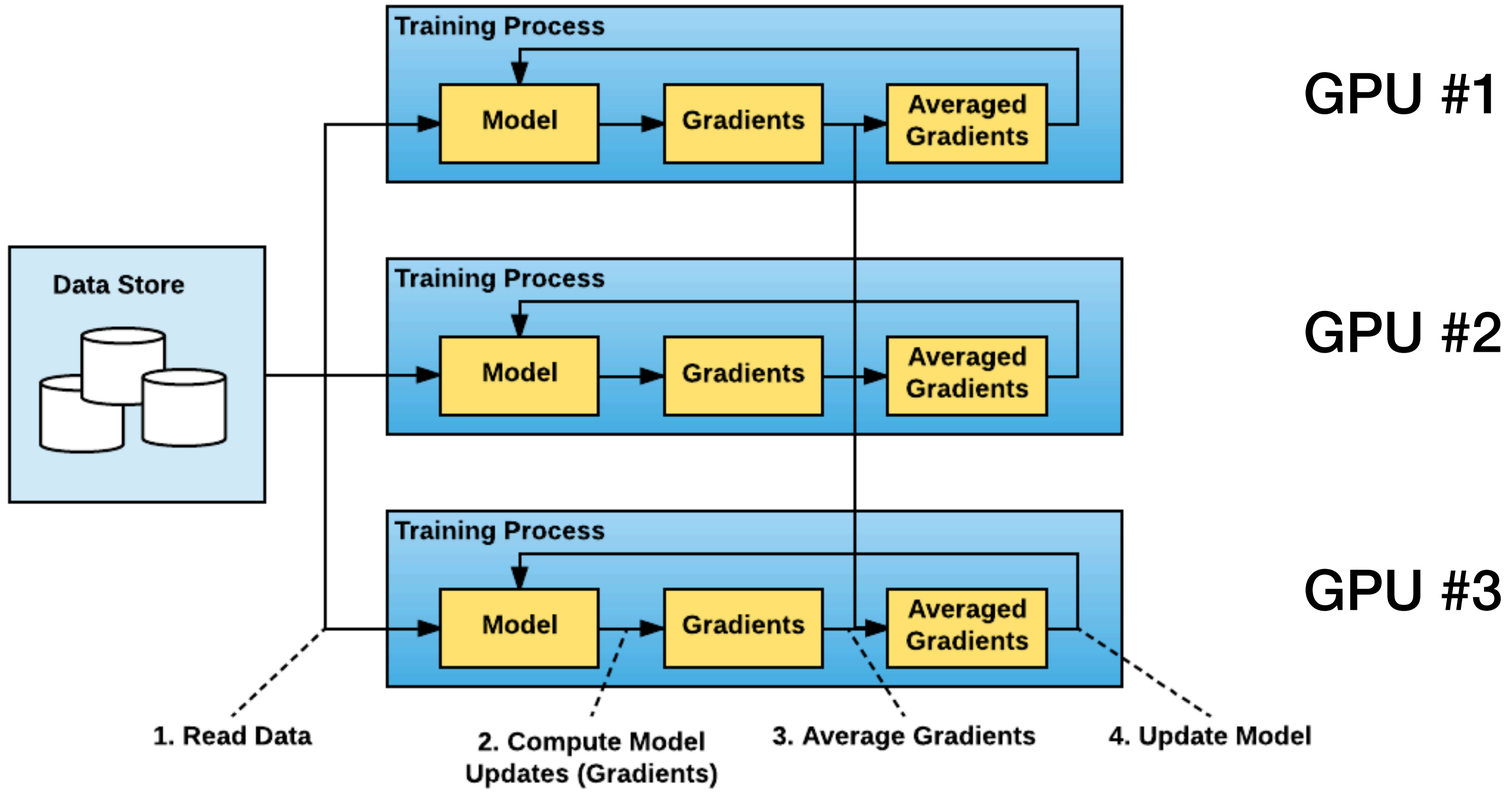
DL Model Training

- Training is a significant bottleneck in DL development.
  - Repeats training steps for a large number of training data.
  - Requires lots of computation & time.



Distributed Training

- Technique to reduce the training time
  by parallelizing the process over multiple GPUs.



Distributed Training in Practice

horovod/**horovod**

Distributed training framework for TensorFlow, Keras, PyTorch, and Apache MXNet.

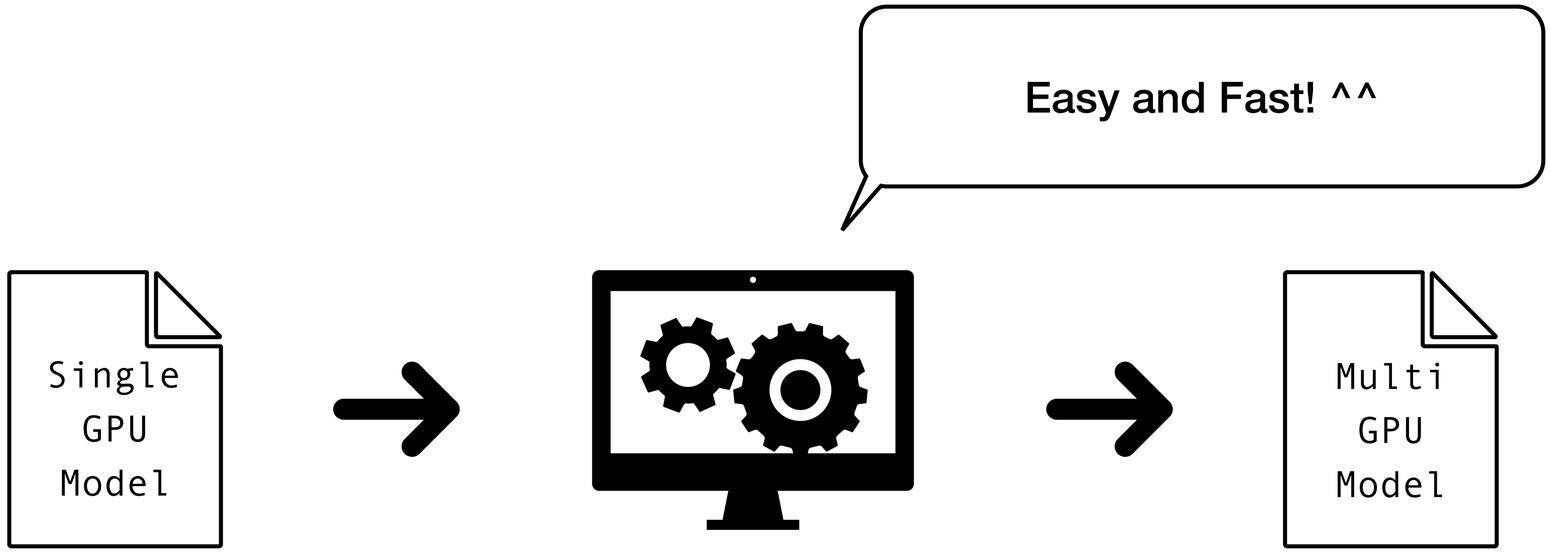
154 Contributors | 603 Used by | 137 Discussions | 13k Stars | 2k Forks

Challenge: Distributing Existing DL models

- Developers have manually rewritten model codes
- The rewriting is time-consuming & labor-intensive.
- Must understand both original DL models & distributed training library.

Solution: Automated Code Transformation for Distributed Training

- Automate the rewriting process with the code transformation technique.



Overview

1. Parse the input model into AST.

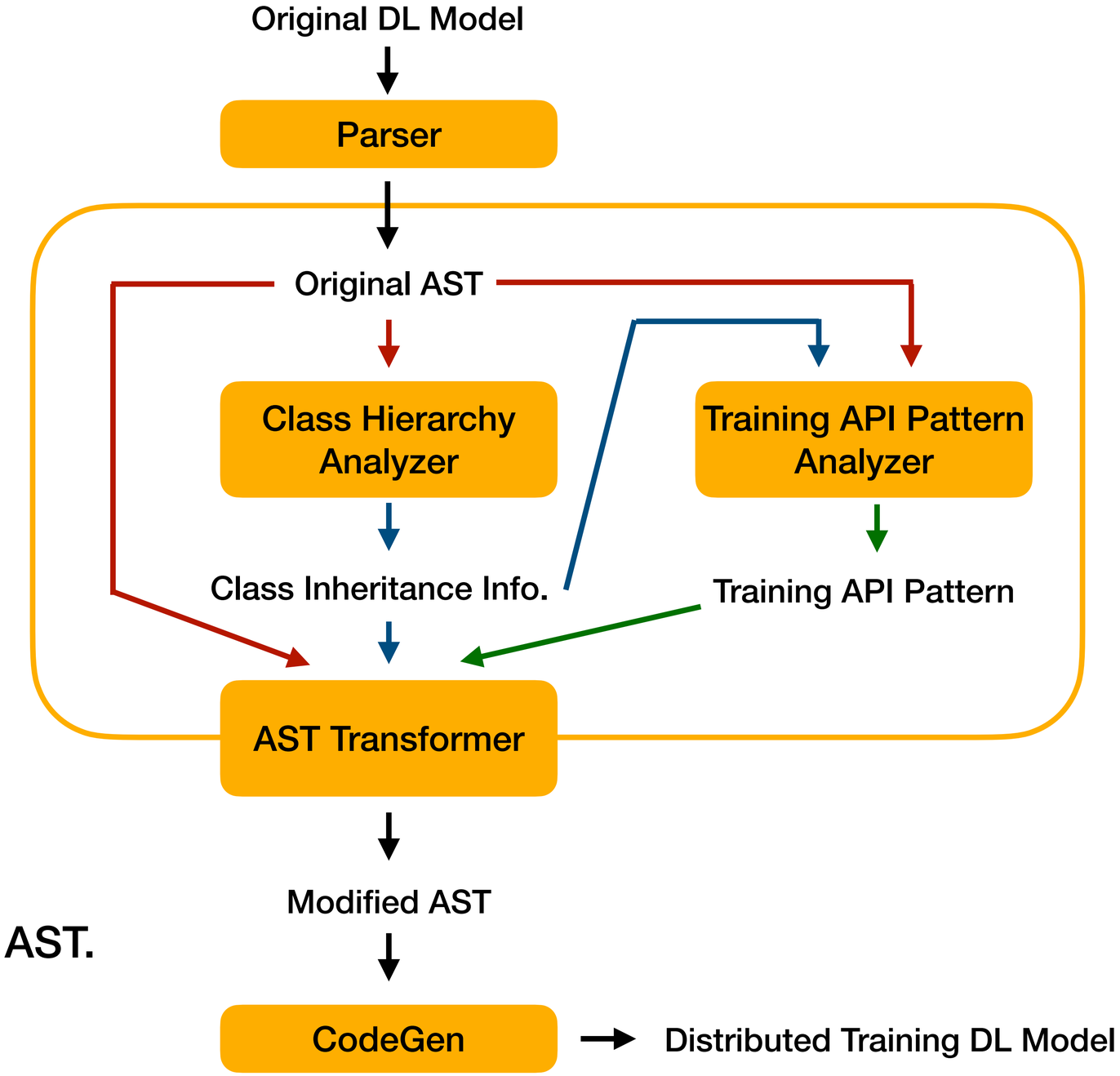
2. Transform the AST.

    2.1. Analyze the class hierarchy.

    2.2. Analyze the training API pattern.

    2.3. Transform the AST,
    using results From 2.1 & 2.2.

3. Generate the distributed model from AST.



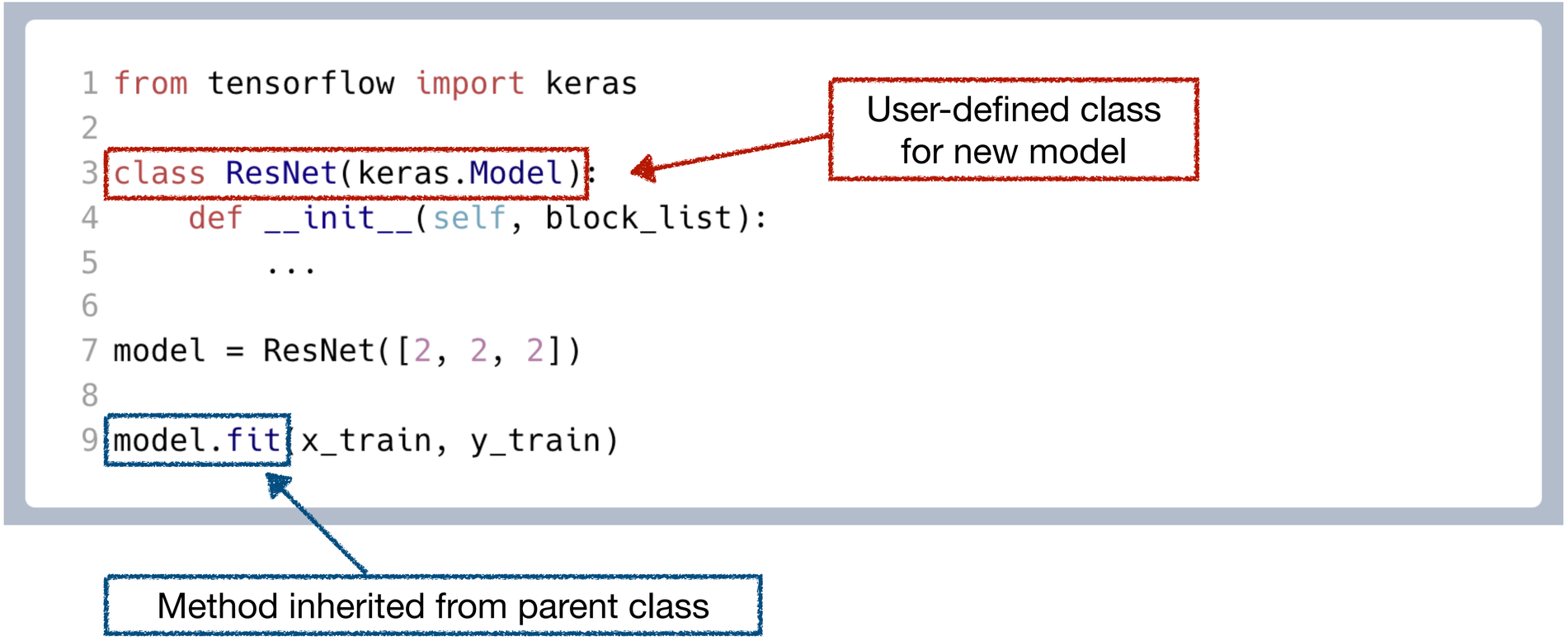
Python Abstract Syntax and Parser

- Full grammar specification in Python Language Reference

module	:=	*stmt*	
stmt	::=	`def` *id* `(`*args*`)` `:` *stmt**	(FUNDEF)
	|	`class` *id* `(`*expr** *keyword**`)` `:` *stmt**	(CLASSDEF)
	|	*expr* `=` *expr*	(ASSIGN)
	|	`for` *expr* `in` *expr* `:` *stmt**	(FORLOOP)
	|	`while` `(`*expr*`)` `:` *stmt**	(WHILELOOP)
	|	`if` `(`*expr*`)` `:` *stmt** (`else` `:` *stmt**)?	(IF)
	|	`with` *with_item** `:` *stmt**	(WITH)
	|	`import` *alias**	(IMPORT)
	|	`from` *i* *id*? `import` *alias**	(IMPORTFROM)
	|	*expr*	(EXPRSTMT)
	|	*stmt* `\n` *stmt*	(SEQUENCE)
expr	::=	*expr* *boolop* *expr*	(BOOLOP)
	|	*expr* *binop* *expr*	(BINARYOP)
	|	*unop* *expr*	(UNARYOP)
	|	`[`*expr**`]`	(LIST)
	|	`(`*expr**`)`	(TUPLE)
	|	*expr* `(`*expr** *keyword**`)`	(CALL)

Class Hierarchy Analysis

- User-defined classes can inherit TensorFlow library classes.
  - Extend or modify behaviors of the classes or methods.



Class Hierarchy Analysis

- Correct transformation rule should identify user-defined classes that inherit training-related library classes.

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

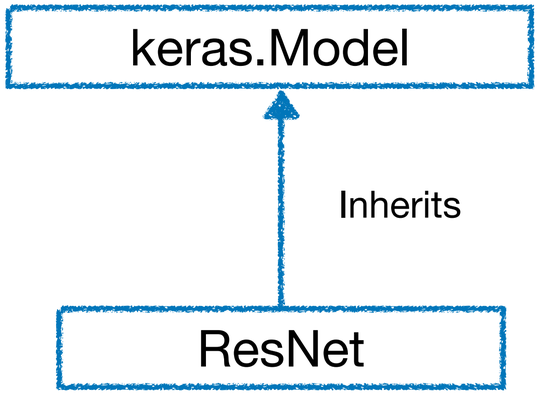
Must know that this instance Inherits ‘fit’ from ‘keras.Model’

Adds argument for the ‘fit’ method

Class Hierarchy Analysis

- Pre-analyze class inheritance relation between TensorFlow library classes and user-defined classes.
- The class hierarchy analysis result is sent to later stages of the transformation.

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



Training API Pattern Analysis

- TensorFlow provides different APIs to define the training.
- Different usage of training APIs requires different transformation rules.

```
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)
```

Repeating training steps with ‘for’ loop

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

Calling ‘fit’ method to
start the automatic training process

Training API Pattern Analysis

- Define four training API patterns categorizing TensorFlow DL models.
- Define a transformation rule for each API pattern.

TF version	Pattern name	Explanation
1.x	Session	Low-level training API using `Session` instance
1.x	MonitoredSession	Low-level training API using `MonitoresSession` instance
2.x	GradientTape	Low-level training API using `GradientTape` instance
2.x	Keras	High-level training API using `fit` method of `keras.models.Model` instance

FIGURE 7 Training API patterns

Session Pattern

```
1 with tf.Session() as sess:
2     for step in xrange(int(num_epochs * train_size) // BATCH_SIZE):
3       sess.run(optimizer, feed_dict=feed_dict)
```

- Use a `Session` instance to manually repeat training steps.
- Search a `with` statement creating a `Session` instance.

MonitoredSession Pattern

```
1 with tf.train.MonitoredTrainingSession(hooks=hooks) as mon_sess:
2     while not mon_sess.should_stop():
3         mon_sess.run(train_op, feed_dict=feed_dict)
```

- Use a `MonitoredSession` instance to manually repeat training steps.
  - Similar to the `Session` pattern.
  - Can add hooks: actions that automatically execute for each step.
- Search a `with` statement that creates a `MonitoredSession` instance.

GradientTape Pattern

```
1 for x, y in train_data.take(training_steps):
2   with tf.GradientTape() as g:
3     pred = conv_net(x)
4     loss = cross_entropy(pred, y)
5
6   trainable_variables = list(weights.values()) + list(biases.values())
7   gradients = g.gradient(loss, trainable_variables)
8   optimizer.apply_gradients(zip(gradients, trainable_variables))
```

- Use a `GradientTape` instance to manually repeat training steps.
  - Record operations inside `with` statement, then automatically compute gradients to optimize model parameters with `apply_gradients` method.
- Search a with statement that creates a `GradientTape` instance.

Keras Pattern

```
1 # model definition
2 class ResNet(tf.keras.models.Model):
3   def __init__(self, params):
4     ...
5
6 model = ResNet([2, 2, 2], num_classes)
7 model.fit(x_train, y_train_ohe, batch_size=batch_size, epochs=epochs,
```

- Define a model with Keras library APIs, and use the `fit` method for automatic training.
- Search for the `Model.fit` method call.
  - Utilize class inheritance information to identify user-defined model classes.

AST Transformation Rule

- Manually inspected Horovod documentation and code examples.
- Define four transformation rules, each for training API patterns.

$$trans_S \;:\; Stmt \rightarrow \Sigma \rightarrow (Stmt\ \textbf{list} \times \Sigma)$$

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

Pattern matching for function argument

Part of formalization of code transformation rule.

Transformation rule for Session pattern

```
1 optimizer = tf.train.MomentumOptimizer(
2     learning_rate = 0.01)
3
4 with tf.Session() as sess:
5     for step in range(num_epochs):
6         sess.run(optimizer, feed_dict)
```

Original Session pattern training code

```
1 optimizer = tf.train.MomentumOptimizer(
2     learning_rate = 0.01 * hvd.size())
3 optimizer = hvd.DistributedOptimizer(optimizer)
4
5 with tf.Session() as sess:
6     for step in range(num_epochs):
7         sess.run(optimizer, feed_dict)
```

Transformed Session pattern training code

- Modify the `Optimizer` instance.
  - Find a statement that assigns the result of `Optimizer` class constructors.
  - Scale the `learning_rate` argument by `hvd.size()`.
  - Wrap the `Optimizer` instance with `DistributeOptimizer()`.

Transformation rule for MonitoredSession pattern

```
1 with MonitoredTrainingSession(hooks=hooks) as mon_sess:
2     while not mon_sess.should_stop():
3         mon_sess.run()
```

Original MonitoredSession pattern training code

```
1 with MonitoredTrainingSession(
2     hooks=hooks.append(hvd.BroadcastGlobalVariablesHook(0)) as mon_sess:
3     while not mon_sess.should_stop():
4         mon_sess.run()
```

Transformed MonitoredSession pattern training code

- Add the hook that broadcasts global variables over multiple processes.
  - Find a with statement that creates a `MonitoredSession` instance.
  - Find (or add) the `hooks` keyword argument.
  - Append the `BroadcastGlobalVariablesHook` to the argument.

Transformation rule for GradientTape pattern

```python
import tensorflow as tf

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

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

Original GradientTape pattern training code

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

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

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

Transformed GradientTape pattern training code

- Modify the `GradientTape` instance.
  - Find a `with` statement that creates a `GradientTape` instance.
  - Wrap the instance with `DistributedGradientTape()`.

Transformation rule for GradientTape pattern

```python
import tensorflow as tf

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

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

Original GradientTape pattern training code

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

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

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

Transformed GradientTape pattern training code

- Add variable broadcasting code after `apply_gradients` method call.
  - Extract the model parameter list from the `apply_gradients` method argument.
  - Add a fixed set of codes after the call.

Transformation rule for Keras pattern

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

optim = keras.optimizers.Adam(0.001)

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

model.fit(x_train, y_train)
```

Original Keras pattern training code

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

optim = keras.optimizers.Adam(0.001 * hvd.size(), )
optim = hvd.DistributedOptimizer(optim, )

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

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

Transformed Keras pattern training code

- Modify the `Optimizer` instance.
  - Find a statement that assigns the result of `Optimizer` class constructors.
  - Scale the `learning_rate` argument by `hvd.size()`.
  - Wrap the `Optimizer` instance with `DistributeOptimizer()`.

Transformation rule for Keras pattern

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

optim = keras.optimizers.Adam(0.001)

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

model.fit(x_train, y_train)
```

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

optim = keras.optimizers.Adam(0.001 * hvd.size(), )
optim = hvd.DistributedOptimizer(optim, )

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

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

- Add the hook that broadcasts global variables over multiple processes.
  - Find the `model.fit` method call.
  - Modify (or add) the `callbacks` keyword argument by adding the `BroadcastGlobalVariablesCallback` hook to the list.

Evaluation: Transformation

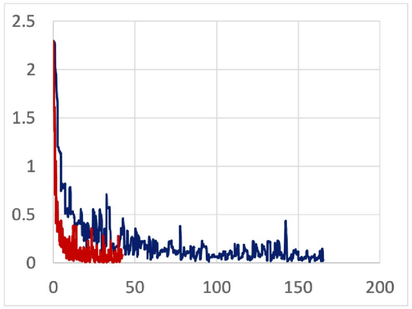
- Experiment on 16 open-source DL models.
- Transformation succeeds on 15/16 models.

Model Name	API Pattern	Transformation Success
LSTM-MNIST	GradientTape	o
SimpleCNN-GradientTape-1	GradientTape	x
SimpleCNN-GradientTape-2	GradientTape	o
SimpleCNN-MonitoredSession	MonitoredSession	o
SimpleCNN-Session	Session	o
VGG-CIFAR10	Keras	o
Play-with-MNIST	GradientTape	o
Linear-Regression	GradientTape	o
Fashion-MNIST	Keras	o
CIFAR10-VGG16	GradientTape	o
Inception-Network	GradientTape	o
RNN-Sentiment-Analysis	Keras	o
Stacked-LSTM-ColorBot	GradientTape	o
Auto-Encoder	GradientTape	o
Variational-Auto-Encoder	GradientTape	o
DCGAN	GradientTape	o

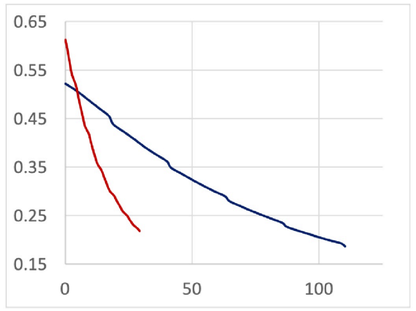
FIGURE 24 Transformation experiment Result

Evaluation: Training

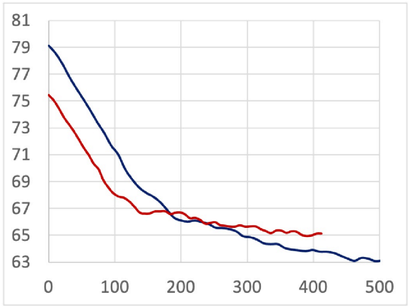
Speed-up: 7/13



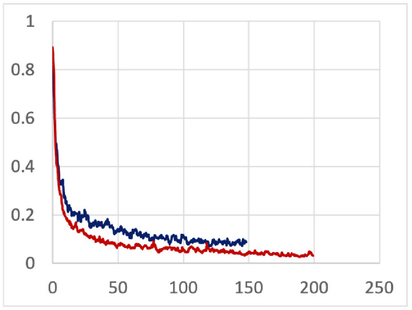
(a) LSTM-MNIST



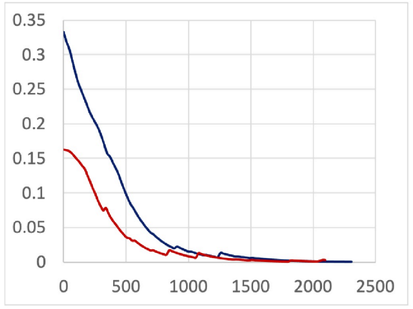
(f) Fashion-MNIST



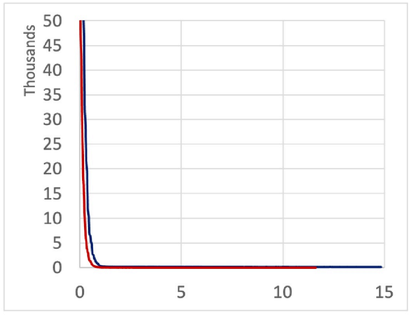
(k) Auto-Encoder



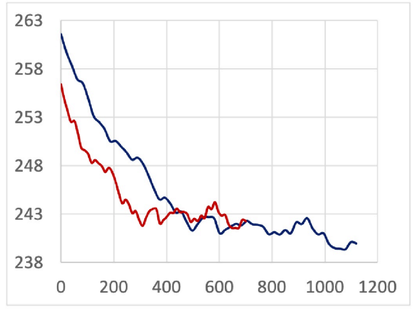
(d) Play-with-MNIST



(h) Inception-network

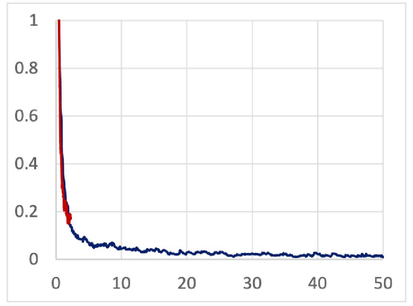


(e) Linear-Regression

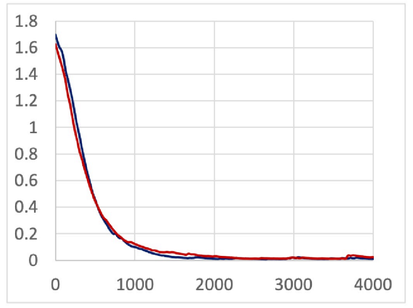


(l) Variational-Auto-Encoder

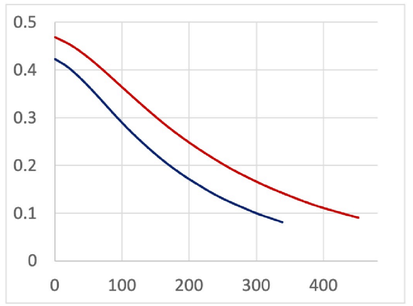
No speed-up or Slower: 3/13



(b) SimpleCNN-GradientTape-2

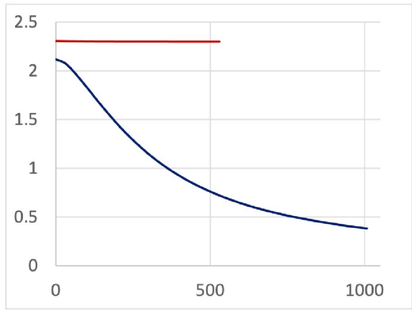


(g) CIFAR10-VGG16

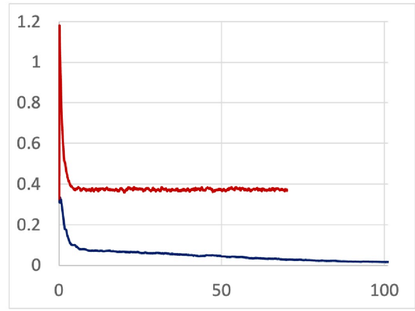


(i) RNN-Sentiment-Analysis

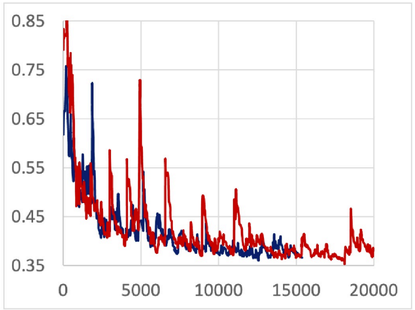
Erroneous Learning: 3/13



(c) VGG-CIFAR10

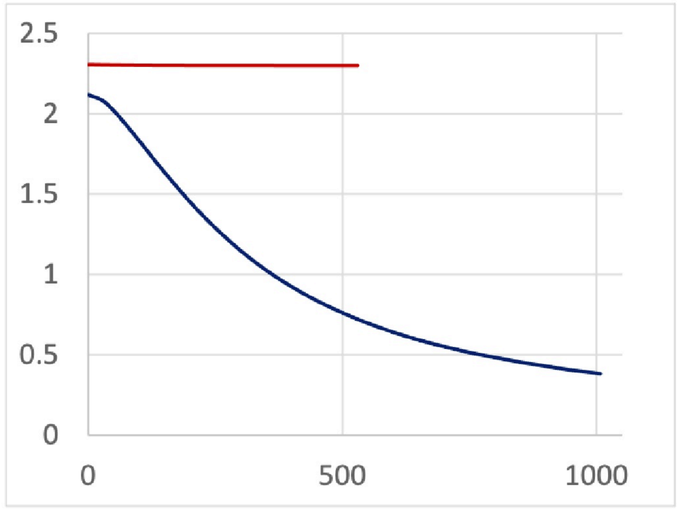


(j) Stacked-LSTM-ColorBot

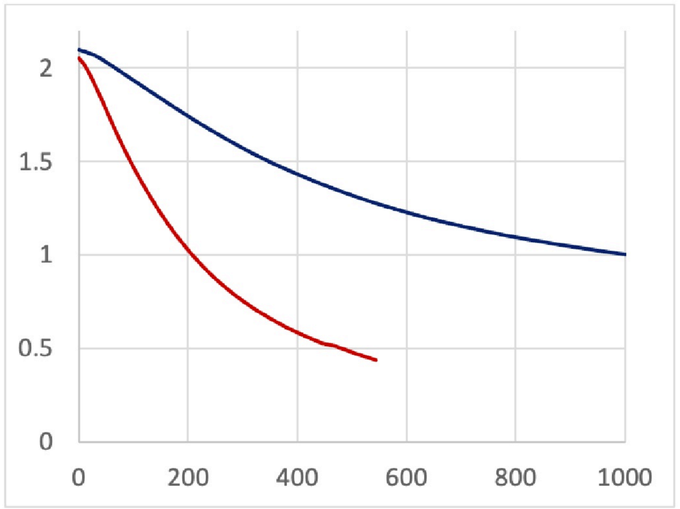


(m) DCGAN

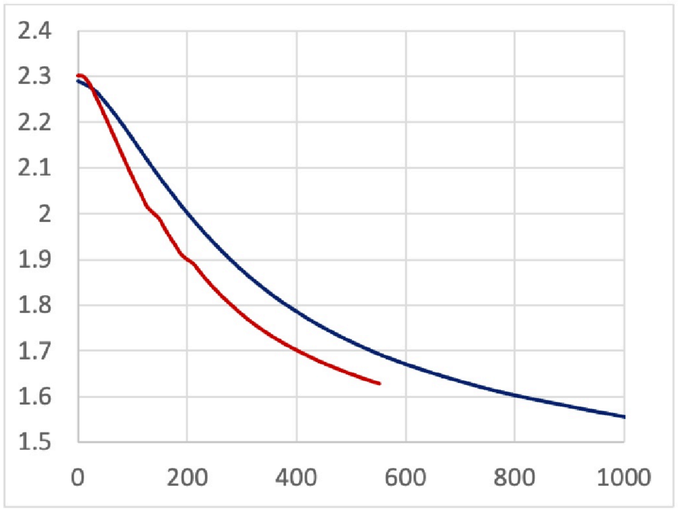
Evaluation: Analysis on Erroneous Case



(a) VGG-CIFAR10 model with lr=0.001 (default value)



(b) VGG-CIFAR10 model with lr=0.0001



(c) VGG-CIFAR10 model with lr=0.00001

"… GPUs provide many parallel processing units which are ideal for throughput computing. However, the design for graphics pipeline lacks some critical processing capabilities for general-purpose workloads, which may result in lower architecture efficiency on throughput computing workloads."

- Lee VW et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU.

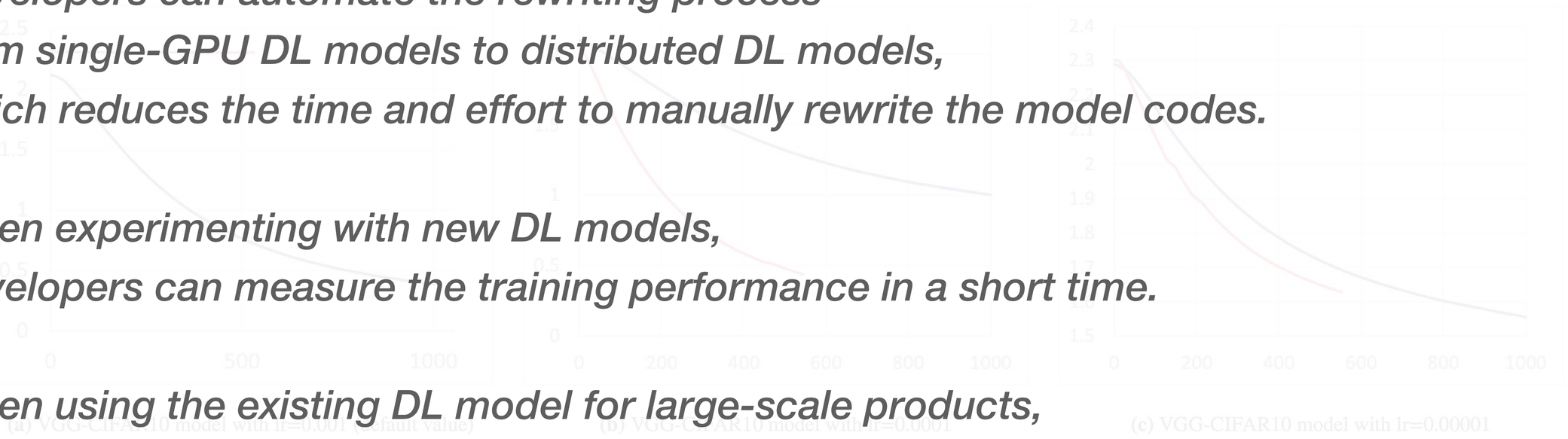
Lee VW, Kim C, Chhugani J, Deisher M, Kim D, Nguyen AD, et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. ACM SIGARCH Computer Architec- ture News. 2010; 38: pp. 451–460. doi: 10.1145/1816038.1816021

Evaluation: Analysis on Erroneous Case

Developers can automate the rewriting process
from single-GPU DL models to distributed DL models,
which reduces the time and effort to manually rewrite the model codes.

When experimenting with new DL models,
developers can measure the training performance in a short time.

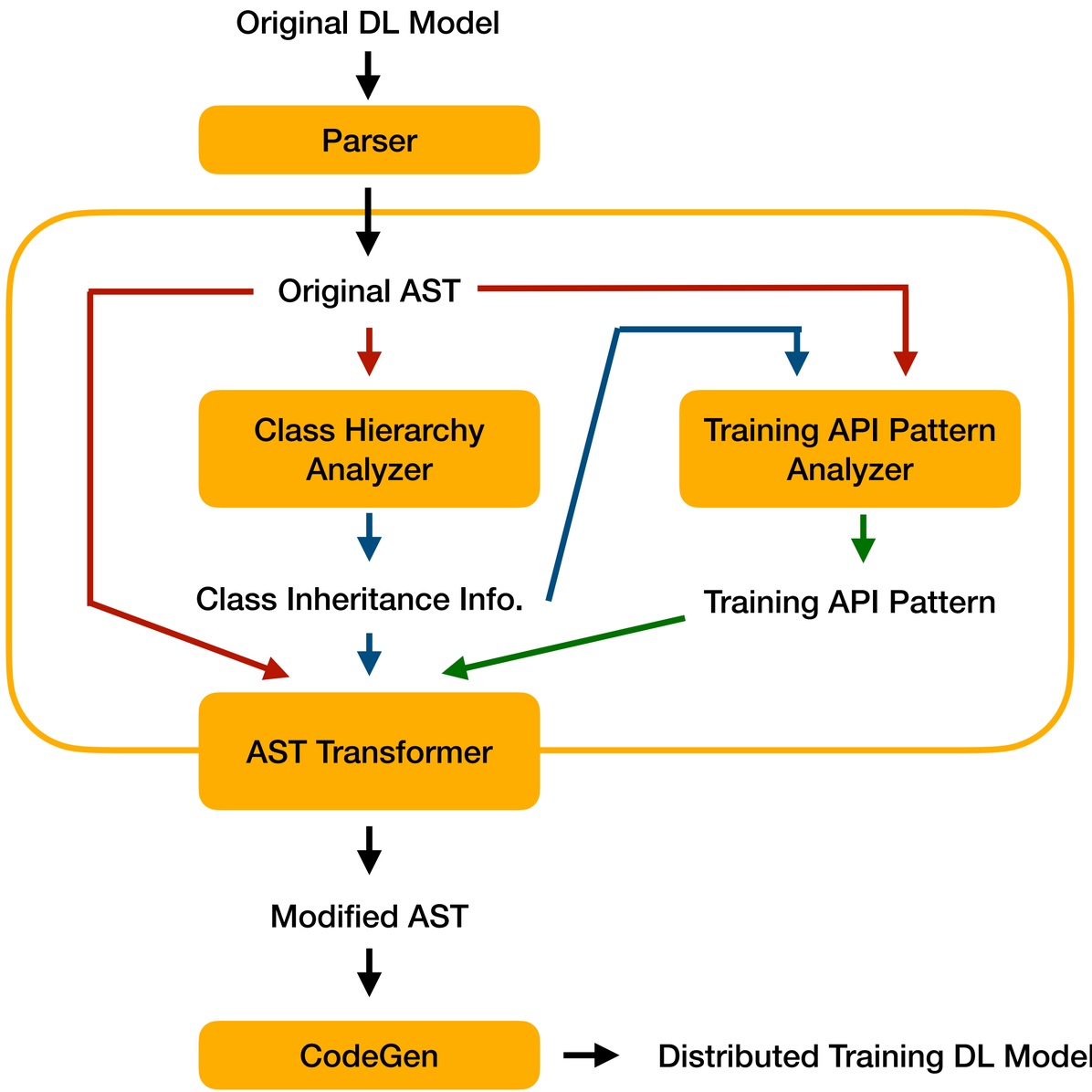
When using the existing DL model for large-scale products,
developers can scale up the model without prior knowledge of the distributed training,
and increase training speed by only modifying training hyperparameters.



"… GPUs provide many parallel processing units which are ideal for throughput computing.
However, the design for graphics pipeline lacks some critical processing capabilities for general purpose workloads, which may result in lower architecture efficiency on throughput computing workloads."

- Lee et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU.

Lee VW, Kim C, Chhugani J, Deisher M, Kim D, Nguyen AD, et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. ACM SIGARCH Computer Architecture News. 2010; 38: pp. 451–460. doi: 10.1145/1816038.1816021



TF version	Pattern name	Explanation
1.x	Session	Low-level training API using `Session` instance
1.x	MonitoredSession	Low-level training API using `MonitoresSession` instance
2.x	GradientTape	Low-level training API using `GradientTape` instance
2.x	Keras	High-level training API using `fit` method of `keras.models.Model` instance

FIGURE 7 Training API patterns