공학석사 학위논문

A Static Analyzer for Detecting Tensor Shape Errors in Deep Neural Network Training Code

심층신경망 학습 코드의 텐서 형상 에러를 찾아내는 정적분석기

2022 년 8 월

서울대학교 대학원

컴퓨터 공학부

주 호 영

A Static Analyzer for Detecting Tensor Shape Errors in Deep Neural Network Training Code

심층신경망 학습 코드의 텐서 형상 에러를 찾아내는 정적분석기

지도교수 허 충 길

이 논문을 공학석사학위논문으로 제출함

2022 년 8 월

서울대학교 대학원

컴퓨터 공학부

주 호 영

주호영의 공학석사 학위논문을 인준함

2022 년 8 월

위 원 장	문병로	(인)
부위원장	허충길	(인)
위 원	이광근	(인)

Abstract

This thesis presents an automatic static analyzer PyTea that detects tensor-shape errors in PyTorch code. The tensor-shape error is critical in the deep neural net code; much of the training cost and intermediate results are to be lost once a tensor shape mismatch occurs in the midst of the training phase. Given the input PyTorch source, PyTea statically traces every possible execution path, collects tensor shape constraints required by the tensor operation sequence of the path, and decides if the constraints are unsatisfiable (hence a shape error can occur). PyTea's scalability and precision hinges on the characteristics of real-world PyTorch applications: the number of execution paths after PyTea's conservative pruning rarely explodes and loops are simple enough to be circumscribed by symbolic abstraction. PyTea is tested against the projects in the official PyTorch repository and some tensor-error code questioned in the StackOverflow. PyTea successfully detects tensor shape errors in these codes, each within a few seconds.

Keywords: Static analysis, deep learning, tensor shape error, SMT solver, Python, PyTorch

Student Number: 2020-29856

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Our Goal

Tensor shape mismatch is a critical bug in deep neural network machine learning applications. Training a neural network is an expensive process that intends to terminate only when it finishes processing a huge amount of data through a sequence of tensor operations. In the middle of this time-consuming training process, if the shape of an input datum failed to fit with a tensor operation, the whole process abruptly stops wasting the entire training cost spent thus far, losing the trained, if any, intermediate result.

Our goal is to automatically predict at compile-time such run-time tensor-shape mismatch errors in PyTorch neural network training code.

1.2 Structure of PyTorch Programs

Contemporary machine learning frameworks such as PyTorch [2], TensorFlow [3], and Keras [4] use Python APIs to build neural networks. Training a neural net-

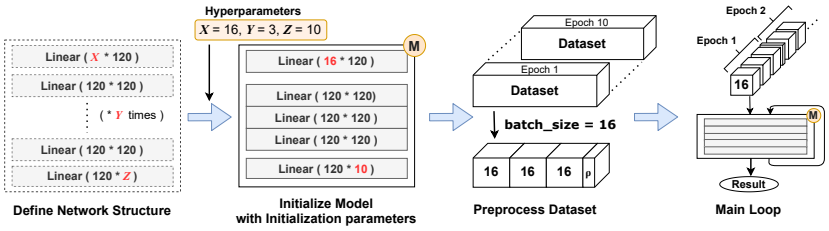

Figure 1.1: Typical structure of neural network training code in PyTorch.

work with such frameworks is mostly patterned after a standard procedure which is illustrated in Figure 1.1. Typical PyTorch neural network training code can be divided into four stages. Figure 1.2 shows a code example, a simplified image classification code taken from the official PyTorch MNIST classification example [5]. We first define the series of neural network layers and make them into a single neural network module. To correctly assemble the layers, the returned tensor of the former layer must satisfy the input requirements of the next layer. We will see those requirements from the next section. The network is instantiated with some initialization parameters called hyperparameter, e.g., the number of hidden layers. Next, the input dataset is preprocessed and adjusted to the requirements of the network. Every dataset is cut into smaller same-sized chunks (called minibatches) from this stage. Finally, the main loop starts, and the minibatches are sequentially fed to the network. One epoch means a single loop that an entire dataset is passed to the network, and the number of epochs (datasets) usually differs depending on the purpose and structure of the neural network. Including the number of epochs, the numbers of iterations in the training code are determined to be constants in most cases, except the main training loop which depends on the size of a dataset.

1.3 Tensor Shape Errors

Figure 1.5 presents the typical type of tensor shape errors, which are slight

```
1  ## 1. DEFINE NETWORK STRUCTURE
2  class Net(nn.Module):
3    def __init__(self, out_classes):
4      super(Net, self).__init__()
5      self.layers = nn.Sequential(
6        nn.Linear(28 * 28, 120),
7        nn.ReLU(),
8        nn.Linear(120, out_classes)
9      )
10
11   def forward(self, x):
12     x = x.reshape(x.shape[0], -1)
13     x = self.layers(x)
14     return x
15
16 ## 2. INITIALIZE MODEL
17 model = Net(out_classes=10)
18
19 ## 3. PREPROCESS DATASET
20 data = dataset.MNIST('./data', train=True,
21     transform=[ToTensor()])
22 loader = DataLoader(data, batch_size=16)
23
24 ## 4. RUN MAIN LOOP
25 for epoch in range(10):
26   for batch, label in loader:
27     # model(batch) == model.forward(batch)
28     output = model(batch)
29     loss = F.nll_loss(output, label)
30     loss.backward()
```

Figure 1.2: Basic PyTorch training code.

modifications of Figure 1.2. From the first example, the second `Linear` layer (line 8), which multiplies the input with 80×10-matrix, requires a specific shape of a tensor as an input. The first layer (line 6), however, returns a wrong-shaped tensor, and the overall pipeline will malfunction. This kind of error is called *tensor shape mismatch* error, simply, shape error.

Shape error is rather hard to manually find, only to be detected by running the program with an actual input. Indeed, the most notorious error for machine learning engineers is the error that can only be occurred after an immense amount

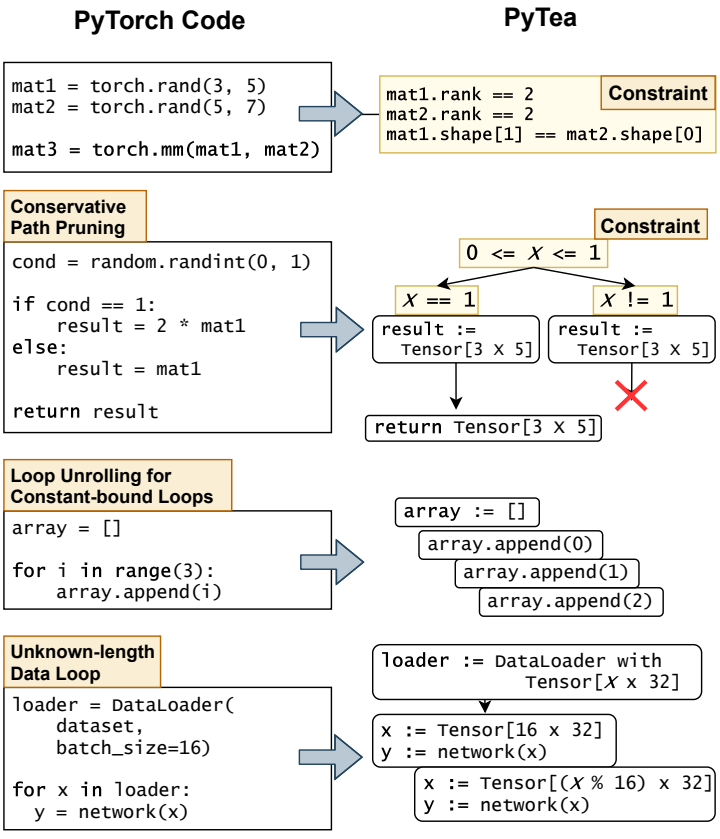

Figure 1.3: Constraint generation example.

of machine-hours.

Figure 1.5(b) shows another example. Its declaration of training data loader (line 14) hides a shape error. `DataLoader` class slices the dataset sequentially by `batch_size` and passes it to the model. If the total length of the dataset is not divisible by `batch_size`, however, the size of the residual minibatch will be the non-zero remainder of the total length. See line 16: because the third parameter `drop_last` is missing, the model assumes a consistent batch size (lines 10 and 6) hence the program will crash from the residual minibatch, losing the whole training hours. The recent massive networks like GPT-3 [6] require more than hundreds of

```python
class RandBlock(nn.Module):
  def __init__(self):
    super(RandBlock, self).__init__()
    self.layer = nn.Linear(32, 32)

  def forward(self, x):
    rand_num = random.randint(0, 1)

    if rand_num == 1:
      result = self.layer(x)
    else:
      result = x

    return result

model = nn.Sequential(
  [RandBlock() for _ in range(24)])
```

Figure 1.4: Path explosion example.

machine-hours to train. This type of error must be noticed before its run.

Figure 1.5(c) illustrates another shape error that can be arisen from a dataset, not a structure of the model. It does not take input from the pre-defined MNIST dataset but reads an image from a file. If the read image is RGB, which has 3×H×W dimensions, it will not fit into the reshape method that requires a tensor of 28×28-elements. That means we have to convert it to a monochrome image before feeding it to the network. Even though it had been successfully tested with monochrome images, there can be a user who tests it with an RGB image, crashing the execution of the code.

Though several works [7, 1, 8, 9, 10] have reported tools to detect the shape mismatch errors of machine learning libraries, especially for TensorFlow [3], none of them have presented any static analysis tool that statically detects the shape errors for realistic Python ML applications. Real-world machine learning applications heavily utilize third-party libraries, external datasets, and configuration parameters, and handle their controls with subtle branch conditions and loops, but

the existing tools still lack in supporting some of these elements and thus they fail to analyze even a simple ML application. To ensure that the shape error will not happen for *any* input data, we should statically infer a precise yet conservative range of each tensor shape and track its transformations through all possible execution paths.

```python
class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.layers = nn.Sequential(
      ## 'B' represents batch size
      ## [B x 784] * [784 x 120] -> [B x 120]
      nn.Linear(28 * 28, 120),
      ## [B x 120] -> [B x 120]
      nn.ReLU(),
      ## [B x 120] * [80 x 10] -> ERROR!
      nn.Linear(80, 10))
```

(a) Error on the network structure.

```python
class Net(nn.Module):
  def __init__(self, batch_size):
    self.batch_size = batch_size
    # ...
  def forward(self, x):
    x = x.reshape(self.batch_size, -1)
    # ...

## some models may require exact batch size
model = Net(batch_size=64)

## POTENTIAL_ERROR 1:
##    argument 'drop_last=True' is essential
loader = DataLoader(data, batch_size=64)
# loader = DataLoader(data, batch_size=64,
#                           drop_last=True)

for epoch in range(10):
  for batch, label in loader:
    out = model(batch)
    ## ERROR ON THE LAST MINIBATCH
    ##     last batch size: 32 (!= 64)
```

(b) Error on the last minibatch.

```python
## POTENTIAL ERROR 2: channel size can be 3
img = PIL.Image.open('./image.png').resize([28, 28])
# img = img.convert('L')

## ERROR WHEN THE IMAGE IS RGB.
tensor = to_tensor(img).reshape(28 * 28)
out = model(tensor)
```

(c) Insufficient data preprocessing.

Figure 1.5: Various type of tensor shape errors.

Chapter 2

Overview of PyTea Analyzer

To find out shape errors before runtime, we present a static analyzer **PyTea** (PyTorch Tensor Error Analyzer). PyTea statically scans PyTorch applications and detects possible shape errors. PyTea analyzes full training and evaluation paths of the Python/PyTorch applications with additional data processing and mixed usage of other libraries (e.g., Torchvision [11], NumPy [12])

Figure 2.1 illustrates the overall architecture of PyTea analyzer. It first translates the original Python codes into a kernel language, PyTea Internal Representation (PyTea IR). Then, it tracks every possible execution path of the translated IR and collects the constraints regarding tensor shapes that dictate the conditions for the code to run without a shape error. The collected constraint sets are given to Satisfiability Modulo Theories (SMT) solver Z3 [13] to judge that those constraints are satisfiable for every possible input shape. Following the result of the solver, PyTea concludes which path contains a shape error or not. If the constraint-solving by Z3 takes too much time, PyTea stops and tells "don't know".

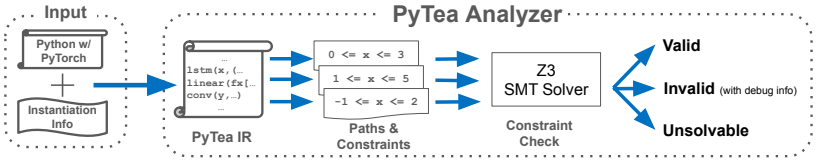

Figure 2.1: Overall architecture of PyTea.

2.1 Assumptions

Given the typical structure of PyTorch neural network training code (Section 1.2), we assume for the PyTea's input the followings about the PyTorch deep neural network training code:

A1 Other than the training or evaluation dataset, every input value required to execute the code is injected by command-line arguments.

A2 There is no infinite loop and recursion. We assume that every loop bound except for the datasets will be fixed to a constant.

A3 The unknown loop bound for the datasets is only for the size of each dataset in an epoch, and every iteration is either with a fixed-sized minibatch of the dataset or with a smaller, residual minibatch.

A4 We assume that string-manipulation expressions have no effect on tensor shapes.

These assumptions are based on our observations that most PyTorch networks and codes can be statically determined to fixed structures once we give precise command-line arguments. Real-world PyTorch applications mostly construct their structures by command-line arguments or external configuration files like JSON files. Therefore, PyTea chooses to analyze programs only with exact command-line arguments.

For a few networks that are not resolved to a single fixed structure, we consider all possible structures. The number of the possible structures is to be controlled by our path-pruning technique, and sometimes, for an inevitable case, by timeout.

2.2 Handling path explosions

The number of possible paths is exponential to the number of branches in sequence. For some complex neural networks, such path explosion is possible. For example, Neural Architecture Search [14] or Networks with Stochastic Depth [15] have branches inside the network themselves. Figure 1.4 shows a representative path explosion case that utilizes a runtime random variable. We can notice that the feed-forward function (`forward(self, x)`) has two execution paths in its body. The final structure of the network is made with 24 same blocks (line 17), which makes 16M paths.

We handle this exponential cost blow-up by means of conservative path-pruning and simple-minded timeouts. If we can find that the result of the binding scope of that feed-forward function is pure (i.e., do not change any global value), and its bounded value is indeed equal for every path and not related with the branch conditions, we then safely ignore other paths except for one. If a path explosion arises even if using this method, we then use a timeout. See Section 3.2.3 for more details.

2.3 Handling Loops

For the loops in typical PyTorch neural network programs, as we discussed in Section 1.2 and accordingly assumed in Section 2.1, we do not need the full power of static analysis [16]. PyTea unrolls constant-bound loops (Assumption A2 in Section 2.1) and analyzes their straight-line code version.

For the unknown-bound loops for datasets, PyTea analyzes the loop body for just two cases with the aforementioned assumption A3. One is for the loop with a fixed-sized regular minibatch of an epoch. The other is for the loop with the residual minibatch. For example, see code in Figure 1.3. For the third code box of Figure 1.3, we can unroll the loop expression to 3 same expressions. If we do not know the length of the dataset, such as the fourth code box of Figure 1.3, we use assumption A3 and consider only two cases for the two different sizes of minibatches.

Chapter 3

Analysis Steps

3.1 PyTea IR

$$
\begin{array}{rcl}
\textit{Expression} & & \\
E & \rightarrow & n \in \mathbb{Z} \mid \texttt{T} \mid \texttt{F} \mid x \ \text{(variable)} \\
 & \mid & \texttt{let}\, x\, E\, E \\
 & \mid & \texttt{if}\, E\, E\, E \\
 & \mid & E \ \textit{bop}\ E \\
 & \mid & \textit{tensor-expr} \\
\textit{bop} & \rightarrow & \textit{numeric-op} \mid \textit{compare-op} \\
\textit{numeric-op} & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \cdots \\
\textit{compare-op} & \rightarrow & \texttt{<} \mid \texttt{=} \mid \cdots \\
\textit{tensor-expr} & \rightarrow & \texttt{mm}\, E\, E \\
 & \mid & \texttt{reshape}\, E\, E\, E \\
 & \mid & \texttt{readImage} \mid \cdots
\end{array}
$$

Figure 3.1: Abstract syntax of PyTea IR.[1]

As the first step of the analysis, the input Python code is translated into the

[1]For the explanatory purpose, we did not include function calls and definitions. See for detailed definitions of PyTea IR. Currently, we implemented 34 basic tensor expressions, and every other PyTorch API has been constructed with the basic expressions. The basic expressions are as

kernel language, *PyTea IR*. See Figure 3.1. PyTorch APIs are translated into tensor expressions that only define shape transformations, which PyTea IR focuses on.

The second step of the analysis is to scan the PyTea IR code and generate constraints.

3.2 Constraint generation

Constraint			
c	$\rightarrow$	$c \wedge c$	
	$\mid$	$c \vee c$	
	$\mid$	$\neg c$	
	$\mid$	e_b	
	$\mid$	$e = e$	
	$\mid$	$e_n < e_n$	
	$\mid$	$\forall \alpha_n \in [e_n, e_n].c$	
Value Expr			
e	$\rightarrow$	$e_s \mid e_n \mid e_b$	
Shape Expr			
e_s	$\rightarrow$	$\texttt{(}\, e_n \texttt{,} \cdots \texttt{,} e_n \texttt{)}$	
	$\mid$	α_s	
	$\mid$	$e_s \texttt{[}\, e_n \texttt{:} e_n \texttt{]}$	
	$\mid$	$e_s \texttt{@}\, e_s$	
Number Expr			
e_n	$\rightarrow$	n	(const number)
	$\mid$	α_n	(unknown number)
	$\mid$	e_n *bop* e_n	(binary operator)
	$\mid$	$\texttt{rank}\,\texttt{(}\, e_s \texttt{)}$	(rank of shape)
	$\mid$	$e_s \texttt{[}\, e_n \texttt{]}$	(e_n-th dim of shape e_s)
	$\mid$	$\prod e_s$	(num of elements in shape e_s)
bop	$\rightarrow$	$\texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \cdots$	
Boolean Expr			
e_b	$\rightarrow$	$\texttt{True} \mid \texttt{False}$	
	$\mid$	α_b	(unknown boolean)
	$\mid$	$e_b \wedge e_b$	(conjunction)
	$\mid$	$e_b \vee e_b$	(disjuction)
	$\mid$	$\neg e_b$	(negation)
	$\mid$	$e = e$	(equality)
	$\mid$	$e_n < e_n$	(less than)

Figure 3.2: Abstract syntax of constraints.

following: `Torch.__init__`, `Torch.__getitem__`, `isSameShape`, `scalar`, `identity`, `broadcast`, `matmul`, `mm`, `bmm`, `item`, `repeat`, `expand`, `expand_as`, `transpose`, `reduce`, `topk`, `view`, `conv2d`, `conv_transpose2d`, `pool2d`, `batchnorm2d`, `cross_entropy`, `cat`, `stack`, `unsqueeze`, `squeeze`, `diag`, `flatten`, `narrow`, `pixel_shuffle`, `layer_norm`, `pad`, `adaptive`, `interpolate`.

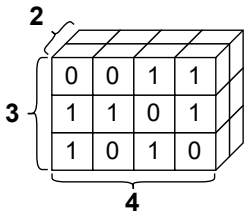

Figure 3.3: A tensor that has shape (2, 3, 4). The rank of this tensor is 3, and each dimension has size 2, 3, and 4.

*Constraint*s are the conditions required by a PyTorch application so that it can be executed without any tensor shape error. For example, two operands of a matrix multiplication operation must share the same dimension. For each tensor operation (`mm`, `reshape`, `readImage`, etc. of Figure 3.1), the shape of the input tensor must obey the requirement of the corresponding operation.

Figure 3.2 shows the abstract syntax of the constraints. *Value expression* represents the value of PyTea IR expressions, which can be used inside shape constraints. When PyTea analyzes a PyTea IR, it traces tensor shapes and primitive values of Python and constructs symbolic value expressions. *Shape expression* represents the shape of tensors, which is basically a tuple of integers $(e_n, \ldots, e_n)$. Figure 3.3 shows an example of a tensor with a shape $(2, 3, 4)$. Each integer is a dimension size. We call the number of dimensions as a *rank* of a shape. We can slice (e_s`[`e_n`:`e_n`]`) a shape expression or concatenate (e_s`@`e_s) two shape expressions. For example, suppose a PyTea IR variable `t` has shape `(2,3,4)`. Expression `t[0]`, which means the first sub-tensor of `t` along the first axis, can be represented inside constraints as `(2,3,4)[1:rank(t)]`, or simply `(3,4)`. In case of expression `t`'s shape is unknown(α_s), the shape of a sub-tensor `t[0]` will be represented as α_s`[1:rank(`α_s`)]`.

3.2.1 Constraint generation rules for PyTea IR

To capture Python semantics and PyTorch shape transformations, PyTea follows the static semantics ($\sigma \vdash E : e, C$) of PyTea IR. Judgment ($\sigma \vdash E : e, C$) means that the PyTea IR expression E is statically approximated by a symbolic value expression e under environment σ in case the constraint set C ($\subseteq$ *Constraint*) is satisfied. The environment σ ($\in$ *Var* $\overset{fin}{\to}$ *Value Expr*) is a finite table that maps variables to symbolic value expressions.

$$\frac{\sigma \vdash E_1 : e, C_1 \qquad \sigma \cup \{x \to e\} \vdash E_2 : e', C_2}{\sigma \vdash \texttt{let}\ x\ E_1\ E_2 : e', C_1 \cup C_2}$$

The introduction of constraints happens for branch expressions or PyTorch APIs (See Section 3.2.2). The other expressions will collect constraints from their subexpressions. For example, for an add expression (E_1+E_2), see:

$$\frac{\sigma \vdash E_1 : e_n, C_1 \qquad \sigma \vdash E_2 : e'_n, C_2}{\sigma \vdash E_1\texttt{+}E_2 : e_n + e'_n, C_1 \cup C_2}$$

The result value is symbolically ($e_n + e'_n$) where e_n and e'_n are symbolic results of E_1 and E_2 respectively. The result constraint set will be a union of the result constraint sets of E_1 and E_2.

Every symbolic variable originates from external input, e.g., random function or a dataset. Every expression in the constraints is constructed by these variables and constant values.

3.2.2 Constraint types

In order to help the constraint resolution engine Z3 come up with a sensible counter-example that violates the derived constraints, we classify the constraints into two exclusive classes: *soft* and *hard* constraints. For Z3 to generate counter-examples, soft constraints can be violated, while hard constraints should not. Thus hard

constraints are, for example, those from branch conditions or about the value range of the input. See Figure 1.3 again. Python built-in `random.randint` function generates an unknown random variable within a given range `[0, 1]`. We mark that bound constraint as a hard constraint. On the other hand, `torch.mm` API demands that two input tensors have to be rank-2 (x, y) tensor and the second dimension (y-coordinate) of the first tensor have to be equal to the first dimension (x-coordinate) of the second tensor. This condition can be violated under the shape of the inputs, hence we mark it as a soft constraint.

Hard constraint generation Hard constraints are those for inputs and branch conditions. Input conditions restrict the initial ranges of each input. Branch conditions split each path into two.

Consider the following rule.

$$\frac{\begin{array}{ll} c_1 = (1 \leq \alpha_n \leq 4) & (new\ \alpha_n) \\ c_2 = (0 < \alpha'_n) & (new\ \alpha'_n) \\ c_3 = (0 < \alpha''_n) & (new\ \alpha''_n) \\ e_s = \texttt{(}\ \alpha_n\texttt{,}\alpha'_n\texttt{,}\alpha''_n\texttt{)} & \end{array}}{\sigma \vdash \texttt{readImage} : e_s, \{c_1, c_2, c_3\}}$$

The `readImage` API is an image fetching API that creates a new 3-rank tensor which represents color channels, height, and width. The range of color channels is from 1 to 4, i.e., monochrome to RGBA, hence the constraint c_1 in the above rule. The symbolic value is a tensor of shape $(\alpha_n, \alpha'_n, \alpha''_n)$.

As another case, consider the following rule.

$$\frac{\begin{array}{ll} \sigma \vdash E_1 : e_1, C_1 & \sigma \vdash E_2 : e_2, C_2 \\ c = (e_1 \leq \alpha_n \leq e_2) & (new\ \alpha_n) \end{array}}{\sigma \vdash \texttt{randInt}\ E_1\ E_2 : \alpha_n, C_1 \cup C_2 \cup \{c\}}$$

The `randInt` API generates a new random variable which is bound to given two numbers. This expression is used from the Python API `random.randint`.

For branching case, see below:

$$\frac{\sigma \vdash E_1 : e_b, C_1 \quad \sigma \vdash E_2 : e, C_2}{\sigma \vdash \texttt{if}\ E_1\ E_2\ E_3 : e, C_1 \cup C_2 \cup \{e_b\}}$$

$$\frac{\sigma \vdash E_1 : e_b, C_1 \quad \sigma \vdash E_3 : e, C_3}{\sigma \vdash \texttt{if}\ E_1\ E_2\ E_3 : e, C_1 \cup C_3 \cup \{\neg e_b\}}$$

The `if` expression creates two paths depending on the branch condition e_b. If the branch condition can be evaluated to a constant boolean, we can safely drop one branch.

Soft constraint generation Soft constraints are the conditions with which PyTorch APIs must comply for them to run without a shape error. For instance, two operands of a matrix multiplication have to share the same middle dimension, and the reshape operation requires that the number of elements of the input tensor must be matched with the number of elements of the target shape. Each PyTorch API holds unique requirements of input conditions, and PyTea collects these requirements as soft constraints.

Following three rules, for example, PyTea collects such constraints from three representative APIs (`mm`, `reshape` and `transpose`):

$$\frac{\begin{array}{c}\sigma \vdash E_1 : e_s, C_1 \quad \sigma \vdash E_2 : e'_s, C_2 \\ \texttt{rank}(e_s) = \texttt{rank}(e'_s) = 2 \\ e''_s = (e_s\texttt{[0]}, e'_s\texttt{[1]}) \quad c = (e_s\texttt{[1]} = e'_s\texttt{[0]})\end{array}}{\sigma \vdash \texttt{mm}\ E_1\ E_2 : e''_s, C_1 \cup C_2 \cup \{c\}}$$

The `mm` API calculates a matrix multiplication of two 2-rank matrices. The second dimension of the first matrix must be equal to the first dimension of the second matrix following the basic rules of linear algebra.

The `reshape` API redefines the shape of a tensor. Reshaping a tensor does not change or drop the value of a tensor, so the target shape must have the exactly

same number of values as the original shape ($\prod e_s = \prod e'_s$):

$$\frac{\begin{array}{cc} \sigma \vdash E_1 : e_s, C_1 & \sigma \vdash E_2 : e_n, C_2 \\ \sigma \vdash E_3 : e'_n, C_3 & e'_s = \texttt{(}e_n\texttt{,}e'_n\texttt{)} \\ \multicolumn{2}{c}{c = (0 < e_n) \wedge (0 < e'_n) \wedge (\prod e_s = \prod e'_s)} \end{array}}{\sigma \vdash \texttt{reshape}\ E_1\ E_2\ E_3 : e'_s, C_1 \cup C_2 \cup C_3 \cup \{c\}}$$

The `transpose` API swaps two dimensions of the tensor E_1 along the E_2-axis and E_3-axis. Unlike the normal 2-rank matrix transposition, `transpose` slices a tensor with (E_2, E_3)-plane and transposes each matrix on each cross-section:

$$\frac{\begin{array}{c} \sigma \vdash E_1 : e_s, C_1 \quad \sigma \vdash E_2 : e_n, C_2 \quad \sigma \vdash E_3 : e'_n, C_3 \\ e_1 = e_s\texttt{[0:}e_n\texttt{]}\ \texttt{@}\ \texttt{(}e_s\texttt{[}e'_n\texttt{])} \quad e_2 = e_s\texttt{[}e_n + 1\texttt{:}e'_n\texttt{]}\ \texttt{@}\ \texttt{(}e_s\texttt{[}e_n\texttt{])} \\ e'_s = e_1\ \texttt{@}\ e_2\ \texttt{@}\ e_s\texttt{[}e'_n + 1\texttt{:rank(}e_s\texttt{)]} \\ c = (0 \leq e_n < e'_n < \texttt{rank(}e_s\texttt{)}) \end{array}}{\sigma \vdash \texttt{transpose}\ E_1\ E_2\ E_3 : e'_s, C_1 \cup C_2 \cup C_3 \cup \{c\}}$$

From this rule, we only consider the shape of the result, not the movement of the value inside the tensor.

3.2.3 Handling path explosion

Splitting execution paths whenever the analyzer encounters a branch can make the analysis cost grow exponentially. We can ignore some of them using the online constraint check, but we cannot for branches that use run-time input values.

However, we can still avoid path split if both paths behave identically in terms of tensor shape. The conservative conditions are as follows:

1. Constraints collected from each path are not dependent on the branch condition, and
2. Each path has no global side-effect, and
3. Two paths' result symbolic values are the same.

PyTea checks the above conditions locally, within the boundary of the `let` expression containing each branch. When PyTea cannot statically decide on any of the three conditions, it safely assumes the conditions do not hold.

Most branches in PyTorch neural network blocks satisfy the above conditions. Typically, network blocks should result in a tensor with a fixed shape that matches with a requirement of the next block or the training target tensor. Those blocks' feed-forward path will be translated into nested `let` blocks with branches that return the same-shaped tensor.

3.3 Constraint check

3.3.1 Online constraint check

To reduce the number of constraints and paths, our analyzer eagerly simplifies the symbolic expressions and constraints with primitive arithmetics and comparisons. By our eager, online constraint check, the ranges of each symbol can sometimes be known and be used to judge the subsequent constraints. If a branch condition can be simplified into constant true or false, we can trace only a single branch without splitting the path. If a constraint can be simplified to constant false, we can immediately report that the path is unsafe.

3.3.2 Offline Constraint check

PyTea feeds the collected constraints of each path to Z3. Algorithm 1 describes how we classify the Z3's result. The final result of PyTea analyzer can be divided into four cases:

- Valid: Soft constraints are always satisfied under the hard constraints. It guarantees that shape error will not occur from this path.

Algorithm 1: Offline Constraint Check with SMT Solver

Input: H, S - logical conjunctions of hard, and soft constraint sets

Output: `valid`, `invalid`, `dontknow`, or `unreachable`

Function $\mathit{analyze}(H, S)$**:**

```
if checkSat(H) = unsat then
    return unreachable
else if S = ∅ then
    return valid
v = checkSat(¬(H → S))
if v = unsat then
    return valid
else if v = sat then
    return invalid
else return dontknow
```

- Invalid: A possible shape error is detected. There is a counterexample that makes soft constraints false under the hard constraints. We also report the generation position of the first broken constraint.
- Don't know: Z3 failed to decide whether constraints are satisfiable or not.
- Unreachable: There is a conflict between hard constraints in this path. In other words, it is impossible to reach this path under the given conditions. This can happen if a path had passed two contradicted branches.

If every path results in either unreachable or valid path, we can conclude that the input program has no tensor shape error.

Chapter 4

Evaluation

Our experiments show PyTea's practical performance for real-world applications. To see the practicality of PyTea, we have collected several complete PyTorch applications and shape-related PyTorch bugs. First, we analyzed the official PyTorch example projects from GitHub repository pytorch/examples[5]. This repository consists 11 complete PyTorch applications about major machine learning tasks from Generative Adversarial Network (GAN) [17] to Natural Language Processing. We also collected some PyTorch shape mismatch errors from StackOverflow and ran PyTea to statically detect them with PyTea. Finally, we conducted case analyses of several fully-functional, hand-made PyTorch applications such as Stochastic ResNet [15].

Experiment Settings PyTea analyzer is written in mainly TypeScript [18], and communicates with Python scripts to run Z3. We also used Pyright [19] to parse and track Python syntax. The experiments were conducted on R7 5800X CPU, node.js 16.0.0 with TypeScript 4.2.4, and Python 3.8.8 with Z3Py 4.8.10.0.

```
└─[$] <git:(main)> python bin/pytea.py --log=1 experiment/examples-bug/dcgan
node /tmp/pytea-paper/bin/index.js experiment/examples-bug/dcgan --logLevel=result-only  --resultPath=/tmp/tmpw_pvirc8/constraint.json
analyzer starts!
──────────── constraint generator result ────────────
<OVERALL: total 2 paths>
potential success path #: 2
potential unreachable path #: 0
immediate failed path #: 0
──────────────── z3 result ────────────────
--- Errornous Path: Path 1 ---
Invalid path: Found conflicted constraints.
  first conflicted constraint (constraint #7):
      message: from 'LibCall.torch.sameShape': not same shapes. at <[231:20 - 231:44] (/tmp/pytea-paper/experiment/examples-bug/dcgan/main.py:231:20)>
  (T[(DataLoader_Batch_I3 * 1)] == T[(DataLoader_Batch_I3 + 1)])
--- Errornous Path: Path 2 ---
Invalid path: Found conflicted constraints.
  first conflicted constraint (constraint #7):
      message: from 'LibCall.torch.sameShape': not same shapes. at <[231:20 - 231:44] (/tmp/pytea-paper/experiment/examples-bug/dcgan/main.py:231:20)>
  (T[(DataLoader_Batch_I3 * 1)] == T[(DataLoader_Batch_I3 + 1)])
<OVERALL: total 2 paths>
  Invalid paths (found conflicted constraints): 2
```

1. Online Constraint Check

Intermediate results

2. Offline Constraint Check with Z3 Solver

Conflicted constraint & reason

Exact code position

Figure 4.1: Test result of PyTea command-line tool.

We fixed the epoch size to 1 from the command-line arguments, but used default values for the other settings. We measured the total elapsed time from the cold boot to the termination of PyTea. The full options and codes are written in the appendix and the external repository. [1]

PyTea command-line tool Figure 4.1 shows an example snapshot of the analysis result of the PyTea command-line tool. It has analyzed one of the PyTorch example projects and prints the result of each phase of PyTea. It first prints out the online constraint check results and categorizes each path into three cases, potential success, potential unreachable, and immediate fail. The last one indicates that the online checker has found a constraint that can be false from that path. The potential unreachable path is the path which the online checker has found a false constraint, but there are certain unresolved branch conditions. That path will be checked at the next phase, and PyTea will examine whether the path has conflicted constraints only within the hard constraint set, which means that the path is unreachable from the beginning.

From the second step, PyTea delivers the collected constraint set of each path

[1] Link: *https://sf.snu.ac.kr/pytea/*

Table 4.1: Analysis result of pytorch/examples code repository. The lines of library APIs encapsulated with the analyzer were counted separately. ○: Analysis succeeded and found injected errors, △: Analysis succeeded but requires a modification of the main code (e.g., provide explicit input tensor), ×: Failed to analyze.

Network	LOC (main + lib)	PyTea	[10]	Total time (s)
dcgan	3714 (214 + 3500)	○	×	1.75
fast_neural_style	4394 (338 + 4056)	○	×	2.40
imagenet	3820 (320 + 3500)	○	×	2.40
mnist	3607 (116 + 3491)	○	×	1.59
mnist_hogwild	3620 (129 + 3491)	○	△	1.94
reinforcement_learning	180 (180 + -)	×	×	-
super_resolution	3886 (193 + 3693)	△	△	1.57
snli	223 (223 + -)	×	×	-
time_sequence_prediction	3333 (88 + 3245)	△	×	1.88
vae	3593 (102 + 3491)	○	△	1.70
word_language_model	3278 (361 + 2912)	△	×	1.81

to Z3 solver and runs the offline constraint checks. The offline check will report the first conflicted constraint and its position of creation, i.e., the exact tensor expression or PyTorch API that causes an error. If the solver does not found any conflicted constraint, PyTea concludes that all the paths are valid, hence no tensor shape error is possible.

4.1 Results

4.1.1 PyTea for PyTorch Examples

For the experiment, we pass each project twice to the analyzer. For the first pass, PyTea analyzed the main code unmodified, and we check that PyTea does not inform false positives. Then, we injected artificial shape errors, which we subtract one from the first dimension of the target tensor, right before the neural network's loss calculation.

This simple method is decided on purpose. From this experiment, we focused on the speed of PyTea which shows the practicallity in order to be integrated to the code editor such as VSCode. This configuration can check the analysis time of the main network, and also confirm that PyTea tracks the tensor operations from the main network thoroughly, and we check PyTea does not report false negative results.

We have compared PyTea against another PyTorch analyzer of Hattori et al. [10]. Table 4.1 shows the overall results. Among the 11 projects, PyTea successfully analyzed 6 projects without any modification of the original source code. For three projects with a complex data preprocessing stage, PyTea needs a bypass (i.e., code modification) of that stage to infer the shapes of input tensors. PyTea has also succeeded in finding these injected errors. As these results show, PyTea is quick and effective enough to be integrated into code editors. Meanwhile, Hattori et al.'s analyzer failed for almost all benchmarks. Furthermore, since their semi-static approach requires an explicit shape of the input tensor, we needed to feed them an exact network model and input tensors to compare its performance with PyTea.

Although we have aimed to analyze the codes without any modification, two projects are heavily dependent on third-party data managing libraries like `OpenAI-Gym`

Table 4.2: Analysis result of the StackOverflow questions. The numbers in parenthesis denote the URL id of each question.

Question	PyTea	[10]
Case 1 (66995380)	○	×
Case 2 (60121107)	○	×
Case 3 (55124407)	○	×
Case 4 (62157890)	○	×
Case 5 (59108988)	○	×
Case 6 (57534072)	○	×

[20]. Because, at the moment, we are focusing on the analysis of PyTorch-centered applications, we decided not to support those libraries for now. Supporting more libraries is straightforward and is our future work.

4.1.2 PyTea for StackOverflow questions

To show that PyTea can identify yet another set of real-world shape mismatches, we collected some PyTorch shape errors from StackOverflow questions. Recent TensorFlow analyzers [1, 8] used a TensorFlow error dataset collected by Zhang et al. [21], but we manually gathered PyTorch shape mismatch cases rather than using their dataset, because of the fundamental difference of the structures between TensorFlow and PyTorch. We also considered porting the TensorFlow error dataset into PyTorch codes, but we concluded that the ported codes are fairly old and artificial and do not reflect the standard method to build a PyTorch application.

Table 4.2 gives the analysis results of the 6 questions that we have collected. PyTea could detect every shape mismatch case from those questions. Following the analysis result, we could find the exact error positions and fix the shape mismatch

```
1  class LSTM(nn.Module):
2    def __init__(self, ...):
3      # 7 lines...
4    def forward(self, tokens):
5      # 5 lines ...
6      return out_scores
7
8  model = LSTM(embedding_matrix=np.zeros((1181, 100)))
9  loss_function = nn.NLLLoss()
10 optimizer = optim.Adam(model.parameters())
11
12 ## CUSTOM INPUT
13 input = torch.ones(256, 4, dtype=torch.long)
14 target = torch.ones(256, 4, dtype=torch.long)
15 output = model(input)
16
17 ## ORIGINAL
18 # output: [256 x 4 x 1181], target: [256 x 4]
19 #     SHAPE MISMATCH: [256 x 1181] != [256 x 4]
20 loss = loss_function(output, target)
21
22 ## FIXED
23 # output: [1024 x 1181], target: [1024]
24 loss = loss_function(output.reshape(256*4, 1181), target.reshape(256*4))
```

Figure 4.2: Example code of StackOverflow question. (Case 2)

cases. For example, the main code (Figure 4.2) of Case 2 does not satisfy the shape conditions for the inputs of `NLLLoss` (line 9). The `NLLLoss` module requires that the shape of the first input tensor without the second dimension is equal to the shape of the second input tensor. PyTea found out that `NLLLoss` could generate a shape error from our experiment. We then fixed the code according to the StackOverflow answer, and PyTea checked that every path became valid.

4.2 Discovered Errors in PyTorch Applications

We applied PyTea to several realistic PyTorch applications which contain potential shape errors or path explosion. PyTea-found shape errors include the typical type of shape errors that we introduced at Section 1.3. The complete projects and

```
1 def load_image(filename, size=None, scale=None):
2   # POTENTIAL ERROR: channel size can be 1.
3   img = Image.open(filename)
4   # img = Image.open(filename).convert('RGB')
5   # ...
6   return img
```

Figure 4.3: Insufficient preprocssing of image file.

experiment scripts from this section are in the external repository.

4.2.1 Detecting insufficient data preprocessing

We found a potential error at the data preprocessing stage from `fast_neural_style` application of pytorch/examples repository. As shown in Figure 4.3, `Image.open` does not guarantee the loaded image has channel 3, i.e., RGB image. Therefore, any training or inference stage with a monochrome image will fail if we miss the channel converting method like line 4. This error was remained from the initial version and was fixed by the latest commit (a3f28a2) of the preprocessing script.

4.2.2 Handling path explosion

For a neural network model which contains a runtime path-explosion, PyTea analyzed it without a timeout. The `stochastic-resnet` example uses several deep learning techniques, mainly stochastic depth training [15]. See Figure 4.4. From this application, the building block of the network contains runtime branches (line 9) that can cause a path explosion. PyTea's path handling algorithm can successfully prune those branches and finishes without timeout. (Caveat: the overall data handling is somewhat hard to follow; we did not automatically reduce the repeat count of the main training loop. We explicitly reduced the length of the dataset (CIFAR-10) with a configuration file (`pyteaconfig.json`), and without modifying the code itself.)

```
def forward(self, x):
  residual = x

  if self.training:
    # sample random float value
    sample = self.m.sample().item()

    ### PATH EXPLOSION
    if sample > 0:
      out = self.conv1(x)
      out = self.bn1(out)
      out = self.relu1(out)
      out = self.conv2(out)
      out = self.bn2(out)

      if self.downsample is not None:
          residual = self.downsample(x)
      out = out + residual
    else:
      if self.downsample is not None:
          residual = self.downsample(x)
      out = residual
  # ...

  out = self.relu2(out)
  return out
```

Figure 4.4: Path explosion in Stochastic ResNet block.

4.2.3 Handling both regular and residual batch sizes in the training loop

PyTea considers a residual minibatch in the training loop which leads to a shape error, as we discussed in Section 2. We simplified the `SimCLR` [22, 23] application to a single PyTorch-only script. From line 4 of Figure 4.5, the main network class `NTXentLoss` takes an exact batch size to initialize itself. So if we omit `drop_last` parameter that removes the last batch at line 32, the last residual minibatch will lead to a crash if the total data size cannot be divided into the batch size. PyTea finds that the inequality between two batch sizes from line 14 of Figure 4.5 generates a shape error.

```
1  class NTXentLoss(torch.nn.Module):
2    def __init__(self, batch_size, temperature):
3      super(NTXentLoss, self).__init__()
4      self.batch_size = batch_size
5      # ...
6    def forward(self, zis, zjs):
7      batch = self.batch_size
8      ## ...
9      ## zis: [B x N], sim: [2B x 2B]
10     ## CONSTRAINT: -sim.shape[0] <= b <= sim.shape[0]
11     l_pos = torch.diag(sim, b)
12     # ...
13     diag = torch.eye(2 * b)
14     l1 = torch.diag(torch.ones(b), -b)
15     l2 = torch.diag(torch.ones(b), b)
16     mask = diag + l1 + l2
17     mask = (1 - mask).type(torch.bool)
18     # 'mask' tensor has (4b^2 - 4b) True values.
19     negatives = sim[mask].view(2 * b, -1)
20     # shape of 'negatives': (2b, 2b - 2)
21     # ...
22 # ...
23 train_loader = DataLoader(train_dataset, batch_size=256,
24   # drop_last=True, # ERROR
25 )
26 losses = train(net, train_loader)
```

Figure 4.5: Shape inference which requires the exact values of a tensor.

4.3 Limitation of PyTea

The main focus of PyTea is the detection of shape errors, so it does not perform general value analysis such as tracking the value of the tensor or array index out-of-bound exception.

If a shape of a tensor is dependent on the value of the other tensor, PyTea can miss a shape error. For instance, the `view` method at line 18 of Figure 4.5 requires that the element count of an input tensor is divisible by $2b$. Tensor masking by a boolean tensor (`similarity_matrix[mask]`) returns a 1-D tensor whose length is equal to the number of `True` of the masking tensor. Although lines 10 to 14 guarantee that the masking tensor has $4b^2 - 4b$ `True`, we do not know the `view`

API will succeed since we do not track the exact value of a tensor.

Chapter 5

Related Works and Conclusion

We have developed an automatic static analyzer PyTea that detects tensor-shape mismatch errors in PyTorch's deep neural network code. Our experiments have shown that PyTea's performance is practical in reality.

Related Works There is only one work [10] of statically detecting shape mismatch of PyTorch applications. Hattori et al. [10] presented a semi-static analysis of PyTorch applications that requires explicit tensor inputs. Because of the path-insensitive and semi-static approach, their tool is premature to fully statically analyze real-world applications. As shown in Table 4.1, the performance of their tool is impractical.

For TensorFlow applications, the latest static analyzer is Pythia [1], following the same group's previous work Ariadne [7]. Pythia is dependent on the Doop framework[24, 25] for Java pointer analysis and the Datalog language. Since Pythia's target is not Python, their coverage of Python and TensorFlow is still insufficient to handle real-world applications. For example, Pythia cannot analyze

```
import tensorflow as tf

target = tf.ones((4, 5))
one, four = 1, 1
if one == 1:
  four = 4

with tf.Session() as sess:
  t0 = tf.ones((3, 4))       # [3 x 4] * [4 x 5]
  p0 = tf.matmul(t0, target) # Pass: Correct
  t1 = tf.ones((3, 5))       # [3 x 5] * [4 x 5]
  p1 = tf.matmul(t1, target) # Error: Correct
  t2 = tf.ones((3, 5 % 2))   # [3 x 2] * [4 x 5]
  p2 = tf.matmul(t2, target) # Pass: False Negative
  t3 = tf.ones((3, 5))[0]    # [5] * [4 x 5]
  p3 = tf.matmul(t3, target) # Pass: False Negative
  t4 = tf.ones((3, 5))[0:1]  # [1 x 5] * [4 x 5]
  p4 = tf.matmul(t4, target) # Pass: False Negative
  t5 = tf.ones((3, four))    # [3 x 4] * [4 x 5]
  p5 = tf.matmul(t5, target) # Error: False Positive
```

Figure 5.1: Basic tensor operations that Pythia [1] fail to analyze correctly.

integer modular operation and tensor indexing and slicing, as shown in Figure 5.1. ShapeFlow [8] is a tester, a dynamic analyzer with fake TensorFlow libraries that only track shape transformations. Their dynamic approach achieved better performance and coverage than Ariadne and Pythia, but it requires a reduced dummy dataset to run their tool. It cannot detect a possibility of shape mismatch caused by an untested input dataset.

There are several works to solve the shape mismatch problems[7, 1, 8, 9, 10], but they all have fundamental limitations to analyze PyTorch machine learning applications, such as the lack of support for handling external data, branches, and loops. Also, most of them work on TensorFlow applications.

PyTorch constructs its graph dynamically which external input value controls the branches and shape of the graph. Any static PyTorch code analyzer has to handle those dynamic semantics. TensorFlow [3] is notoriously hard to debug as it

constructs the networks statically; their development team had decided to change its basis to dynamic construction like PyTorch framework in TF version 2.0. The migration to 2.0 means it will outdate the previous TensorFlow analyzers, and we expect that our work can be adapted to TF 2.0.

Static analyses for Python programs have also been reported [26, 9]. Notably, Cruz-Camacho's thesis[9] contains the shape analysis of NumPy[12] array operators. However, their coverage of Python syntax is restricted that custom function and class declaration are not supported. PyExZ3[27] is a value analyzer for Python language that implemented dynamic symbolic executor with Z3 backend. To port it for a shape mismatch problem needs a sizeable overhaul.

Appendix A

Appendix

A.1 Supported Python syntax

PyTea's Python parsing stage is dependent on the Pyright type checker. From the Python syntax supported by Pyright, we can analyze these statements or experiments below:

- Assignment, Member access, Indexing
- Unary/Binary operation, operator overloading
- `if ...: ... elif ...: ... else: ...`
- `for ... in ...: ...`
- `break`, `continue`, `return`, `pass`
- Function/Closure definition
- Function call
- Variadic, keyword arguments (`*args, **kwargs`)
- Class (single-inheritance)
- `__getitem__`, `__init__`, `__call__`

- `global`, `nonlocal`
- `lambda`, ternary operator (`... if ... else ...`)
- Tuple, List, Dictionary initialization
- List comprehension, List/Dictionary unpacking, List slicing
- Import local script

The statements below are unsupported by PyTea. These statements will be ignored.

- Augmented assignment
- `async`, `await`
- `for ...: ... else: ...`
- Custom `__getattr__`, `__setattr__`, `__setitem__`
- Class (multiple-inheritance)
- Decorator, `@staticmethod`, `@classmethod`
- Generator (`yield`)
- Formatted string (e.g., `f"{...}"`)
- Keyword/Positional-only parameters (e.g., `def f(x, /, y, *, z)`)
- Set, Frozen set
- Type annotation
- Import 3rd-party (pip) library
- Side-effects (File I/O, Networking, ⋯)

The statements below can be parsed or analyzed, but we do not guarantee their analysis are correct. We will describe their behavior in PyTea.

- `while`: PyTea assumes that every loop is finite. The maximum iteration counts will be bound to 300.
- Iterator protocol (`__iter__`, `__next__`): Support of iterator protocol is still premature. Because of the finite loop assumption, an iterator instance should have constant length too. (i.e., iterator should implement `__len__`)

- `raise`: Exception handling is not supported. If an exception is raised, the analyzer will be terminated with an error.
- `try: A except ...: ... else: ... finally: B`: This will be translated to `A; B`
- `with A: B`: This will be translated to `A; B`
- `assert ...`: If assert condition is definitely false, PyTea will report an error without handling exception.
- `del ...`: Only removing variable is supported.

For the Python builtin and 3rd-party libraries, See `bin/dist/pylib` directory.

A.2 Evaluation details

From the Section 4, we measured the mean analysis time of each project in pytorch/examples repository. Analysis time means the total execution time of `bin/pytea.py`. Each project is analyzed 5 times first for warmup. Then, we measured the mean running time of 30 repeated analyses.

The actual analysis time (i.e., online checker and Z3 execution time) will be shorter than the experiment results because the total execution time includes the interpreting time of 1.2MB JavaScript codes, importing time Z3 runtime, and translation time of PyTorch implementation of PyTea.

Use `-l=2` options (e.g., `python bin/pytea.py -l=2 path/to/script.py`) for more detailed time measurement. It will show its execution times step by step. We expect that subsequent optimizations like caching translated PyTorch IR will reduce most of the execution time.

A.2.1 Specification of injected shape error

We modified the shape of the target tensor from the loss functions. From the projects in pytorch/examples, every training loop ends with loss calculation using

loss modules or functions like `torch.nn.NLLLoss`. Those functions require two inputs, target value and predicted value made by neural networks. We subtracted one from the first dimension of the target tensor using the expression like `target[1:]`.

This simple method is decided on purpose. From this experiment, we focused on the speed of PyTea which shows the practicallity in order to be integrated to the code editor such as VSCode. Also, there are not much common feature between the diverse projects. To check the overall speed of PyTea, we choose to inject the shape error right after the feed-forward path of the main network. This configuration can check the analysis time of the main network.

We can check that PyTea really traced every function and method inside the main network. We have implemented precise warning messages which notifies to user if PyTea met an unimplemented API. PyTea reports no errors and few unimportant warnings if we do not inject shape error, and only reports an error if we inject an error. The user also can check PyTea's tracing by fixing the class definition of the network model.

A.2.2 Analysis result of complete PyTorch project

We included three complete PyTorch projects: `SimCLR`, `stochastic-resnet`, and `transformer`. The last project is not introduced from the experiment results. The `transformer` project is simplified version of huggingface/transformer repository [28] which is the collection of several Natural Language Processing methods and networks such as BERT model [29]. We injected a shape error at line 181 of `modeling_bert.py`. PyTea can successfully find that error. To see the complete analysis results of PyTorch projects, run `test_all.sh`.

Table A.1: The list of complete command-line arguments

Network	Command-line arguments
dcgan	`--cuda --dataset=cifar10` `--niter=1 --manualSeed=1`
fast_neural_style	`train --epochs=1 --cuda=1` `--dataset=./ --save-model-dir=./`
imagenet	`--data=./data --gpu=1 --epochs=1`
mnist	`--epochs=1`
mnist_hogwild	`--data=./data --epochs=1`
super_resolution	`--cuda --upscale_factor=2 --nEpochs=1`
time_sequence_prediction	`--steps=1`
vae	`--epochs=1 --seed=1`
word_language_model	`--epochs=1 --cuda`

A.2.3 Complete command-line arguments

Basically, we use default command line arguments except for epoch count and GPU usage. However, we have to give an explicit value to some arguments in which `required=True` is set. Some projects are given explicit seeds, but it does not affect the main evaluation. Table A.1 shows the full command-line arguments used from the evaluation. See `pyteaconfig.json` in each project.

A.2.4 Code modification points

As we mentioned in the Section 4, some codes need bypasses of the complex data processing stage. From the pytorch/examples projects, we commented out some lines that perform external text file manipulation. Those points are marked with a comment `# <FIXED LINE FOR EXPERIMENT>`.

StackOverflow users usually upload their codes as a code snippet which cannot be solely executed. Therefore, we added some basic training or network initialization codes to be able to analyzed by PyTea.

A.2.5 Experiment comparison criteria

We have compared with the PyTorch analyzer of Hattori et al. [10]. Since their semi-static approach needs an explicit network and input tensor, we followed their testing code [30] and made each testing model at `experiment/elichika`. Two StackOverflow questions' (UT-1 and UT-3) bugs are not from a model configuration (inconsistent batch size and misuse of API, respectively). We did not test them on Hattori et al.

We tested their tool with `test_elichika.py`. If the analysis of a network shows a warning which failed to infer the type of some module or returns symbolic value or dimension None, we mark those tests as failed.

A.3 Complete definitions of PyTea IR syntax and semantics

A.3.1 Syntax

$E \in$ *expr*	$\rightarrow$	$n \in \mathbb{Z} \mid r \in$ *Float* $\mid$	
		$s \in$ *String* $\mid$ `True` $\mid$ `False` $\mid$ `None`	(constants)
	$\mid$	`Object`	(new object)
	$\mid$	`Tuple(`*expr*$^+$`)`	(tuple)
	$\mid$	`Call(`*expr*, *expr**`)`	(function call)
	$\mid$	`LibCall(`*id*, *expr**`)`	(library function call)
	$\mid$	`BinOp(`*binary-op*, *expr*, *expr*`)`	(binary operation)
	$\mid$	`UnaryOp(`*unary-op*, *expr*`)`	(unary operation)
	$\mid$	*left-val*	(assignable)
$E_l \in$ *left-val*	$\rightarrow$	`Name(`*id*`)`	(name node)
	$\mid$	`Attribute(`*expr*, *id*`)`	(attribute)
	$\mid$	`Subscript(`*expr*, *expr*`)`	(subscript)
$S \in$ *stmt*	$\rightarrow$	`Pass`	(skip)
	$\mid$	`Expr(`*expr*`)`	(evaluation)
	$\mid$	`Seq(`*stmt*, *stmt*`)`	(sequence)
	$\mid$	`Assign(`*left-val*, *expr*`)`	(assignment)
	$\mid$	`If(`*expr*, *stmt*, *stmt*`)`	(conditional)
	$\mid$	`ForIn(`*id*, *expr*, *stmt*`)`	(bounded loop)
	$\mid$	`Return(`*expr*`)` $\mid$ `Continue` $\mid$ `Break`	(control flow)
	$\mid$	`Let(`*id*, (*expr* $\mid$ `Undef`), *stmt*`)`	(variable def)
	$\mid$	`FunDef(`*id*, *id**, *stmt*, *stmt*`)`	(function def)
binary-op	$\rightarrow$	*numeric-op* $\mid$ *compare-op* $\mid$	
		bool-op $\mid$ *list-op*	
numeric-op	$\rightarrow$	`+` $\mid$ `-` $\mid$ `*` $\mid$ `**` $\mid$ `/` $\mid$ `//` $\mid$ `%`	
compare-op	$\rightarrow$	`<` $\mid$ `<=` $\mid$ `==` $\mid$ `!=`	
bool-op	$\rightarrow$	`and` $\mid$ `or` $\mid$ `is` $\mid$ `is not`	
list-op	$\rightarrow$	`in` $\mid$ `not in`	
unary-op	$\rightarrow$	`not` $\mid$ `-`	

`LibCall` expression is used to implement some complex Python syntax (e.g., keyword argument) and *tensor-expr* from the main paper. Also, some complicated behaviors are implemented with a single `LibCall` for optimization purposes. Every other syntax is essentially the same as the corresponding one of Python.

A.3.2 Constraint

Constraint	c	$\rightarrow$	$c \wedge c$	
		$\mid$	$c \vee c$	
		$\mid$	$\neg c$	
		$\mid$	e_b	
		$\mid$	$e = e$	
		$\mid$	$e_n < e_n$	
		$\mid$	$\forall \alpha_n \in [e_n, e_n].c$	(c is true forall α_n)
		$\mid$	`broadcastable` (e_s, e_s)	
Value Expr	e	$\rightarrow$	$e_s \mid e_n \mid e_b \mid e_{str}$	
Shape Expr	e_s	$\rightarrow$	(e_n , ... , e_n)	(ranked shape)
		$\mid$	α_s	(unknown shape)
		$\mid$	e_s [e_n : e_n]	(shape slicing)
		$\mid$	e_s @ e_s	(shape concat)
		$\mid$	`setdim` (e_s, e_n, e_n)	(set e_nth dim of shape e_s to e_n)
		$\mid$	`broadcast` (e_s, e_s)	(broadcast shapes)
Number Expr	e_n	$\rightarrow$	$n \in Int + Float$	
		$\mid$	α_n	(unknown number)
		$\mid$	$uop\ e_n$	
		$\mid$	$e_n\ bop\ e_n$	
		$\mid$	`max` (e_n^+) $\mid$ `min` (e_n^+)	
		$\mid$	`rank` (e_s)	(rank of shape e_s)
		$\mid$	e_s [e_n]	(e_nth dim of shape e_s)
		$\mid$	$\prod e_s$	(numel of shape e_s)
Boolean Expr	e_b	$\rightarrow$	`True` $\mid$ `False`	
		$\mid$	α_b	(unknown boolean)
		$\mid$	$e_b \wedge e_b$	
		$\mid$	$e_b \vee e_b$	
		$\mid$	$\neg e_b$	
		$\mid$	$e = e$	
		$\mid$	$e_n < e_n$	
String Expr	e_{str}	$\rightarrow$	$s \in String$	
		$\mid$	α_{str}	(unknown string)
		$\mid$	e_{str} [e_n : e_n]	(string slicing)
		$\mid$	e_{str} @ e_{str}	(string concat)
Binary Operation	bop	$\rightarrow$	`+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `//` $\mid$ `%`	
Unary Operation	uop	$\rightarrow$	`neg` $\mid$ `floor` $\mid$ `ceil` $\mid$ `abs`	

`broadcastable` and `broadcast` follows the numpy broadcasting rule. You can see the broadcasting rule from `https://numpy.org/doc/stable/user/basics.broadcasting.html`.

A.3.3 Domain

$$
\begin{array}{lcrcl}
\sigma & \in & \mathit{Env} & = & \mathit{Id} \overset{fin}{\to} \mathit{Addr} \\
\mathcal{H} & \in & \mathit{Heap} & = & \mathit{Addr} \overset{fin}{\to} \mathit{Value} \\
v & \in & \mathit{Value} & = & \mathit{Value}\ \mathit{Expr} + \mathit{Addr} + \mathit{Object} \\
 & & & & \quad + \mathit{Func} + \texttt{\{ None, NotImpl, Undef \}} \\
o & \in & \mathit{Object} & = & (\mathit{Id} + \mathit{Int} + \mathit{String}) \overset{fin}{\to} \mathit{Value} \\
f & \in & \mathit{Func} & = & \mathit{Id} \times \mathit{Id}^* \times \mathit{Stmt} \times \mathit{Env} \\
\kappa & \in & \mathit{Cont} & = & \texttt{\{run, cnt, brk\}} \\
l & \in & \mathit{Addr} & & \text{(address space)} \\
i, x & \in & \mathit{Id} & & \text{(identifier)} \\
C & \in & 2^{\mathit{Constraint}} & & \text{(constraint set)}
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{"length"}, \texttt{""}\text{(empty string)}, \ldots & \in & \mathit{String} \\
\_\_\mathit{len}\_\_, \_\_\mathit{add}\_\_, \_\_\mathit{getitem}\_\_, \ldots & \in & \mathit{Id}
\end{array}
$$

From the following semantics, the domain type of *Object* can be distinguished by its font. Domain *String* is used for dictionary and `__getitem__` method (e.g., `obj["key"]`). *Id* is used for object attribute and `__getattr__` method (e.g., `obj.key`). *Int* is used for list and `__getitem__` method (e.g., `obj[0]`).

Every Python object such as class, list, tuple, dictionary will be translated to *Object*. For optimization purpose, the actual implementations of *Object* and *Func* have some additional information such as function default parameter.

```
class A:      # A.__mro__ = (A, object)
  pass
class B(A):   # B.__mro__ = (B, A)
  pass
b = B()       # b.__mro__ = (B, A)
```

Figure A.1: Class and instance ___*mro*___.

There are two reserved attributes for each *Object* value. *$length* indicates the length of object if it is a lengthed value like list. ___*mro*___ indicates the Method Resolution Order (MRO) of Python. PyTea assigns a two-element tuple to attribute ___*mro*___ of each object. The first element is its class if the object is instance,

or itself if the object is a class. The next object is the first element's parent class. Unlike the original Python, PyTea does not support multiple inheritance, that means the set of $\_\_\_mro\_\_\_$ forms the edges of class inheritance tree.

Bibliography

[1] S. Lagouvardos, J. Dolby, N. Grech, A. Antoniadis, and Y. Smaragdakis, "Static Analysis of Shape in TensorFlow Programs," in *34th European Conference on Object-Oriented Programming*, ECOOP 2020, (Dagstuhl, Germany), pp. 15:1–15:29, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, " TensorFlow: Large-Scale Machine Learning on Heterogeneous

Systems." https://www.tensorflow.org/, 2015. Software available from tensorflow.org.

[4] F. Chollet *et al.*, "Keras." `https://keras.io`, 2015.

[5] "pytorch/examples." https://github.com/pytorch/examples, 2017.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners." https://arxiv.org/abs/2005.14165, 2020. arXiv:2005.14165.

[7] J. Dolby, A. Shinnar, A. Allain, and J. Reinen, "Ariadne: Analysis for Machine Learning Programs," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, (Philadelphia, PA, USA), p. 1–10, Association for Computing Machinery, 2018.

[8] S. Verma and Z. Su, "ShapeFlow: Dynamic shape interpreter for TensorFlow." https://arxiv.org/abs/2011.13452, 2020. arXiv:2011.13452.

[9] E. Cruz-Camacho, "Static Analysis of Python Programs using Abstract Interpretation: An Application to Tensor Shape Analysis," Master's thesis, Universidad Nacional de Colombia - Sede Bogotá, 2019.

[10] M. Hattori, S. Sawada, S. Hamaji, M. Sakai, and S. Shimizu, "Semi-Static Type, Shape, and Symbolic Shape Inference for Dynamic Computation Graphs," in *Proceedings of the 4th ACM SIGPLAN International Workshop*

on Machine Learning and Programming Languages, MAPL 2020, (London, UK), p. 1119, Association for Computing Machinery, 2020.

[11] "Torchvision." https://github.com/pytorch/vision, 2016.

[12] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature,* vol. 585, pp. 357–362, Sept. 2020.

[13] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, (Budapest, Hungary), p. 337–340, Springer-Verlag, 2008.

[14] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *5th International Conference on Learning Representations*, (Toulon, France), OpenReview.net, 2017.

[15] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, "Deep networks with stochastic depth," in *14th European Conference on Computer Vision*, (Amsterdam, Netherlands), pp. 646–661, Springer, 2016.

[16] X. Rival and K. Yi, *Introduction to Static Analysis: An Abstract Interpretation Perspective.* Cambridge, MA, USA: The MIT Press, 2020.

[17] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," in *4th Inter-*

national Conference on Learning Representations, (San Juan, Puerto Rico), 2016.

[18] "Typescript." https://www.typescriptlang.org/, 2012.

[19] "Pyright." https://github.com/microsoft/pyright, 2018.

[20] "OpenAI Gym." https://github.com/openai/gym, 2016.

[21] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An Empirical Study on TensorFlow Program Bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, (Amsterdam, Netherlands), p. 129–140, Association for Computing Machinery, 2018.

[22] T. Chen, S. Kornblith, M. Norouzi, and G. E. Hinton, "A simple framework for contrastive learning of visual representations," *CoRR*, vol. abs/2002.05709, 2020.

[23] "sthalles/ SimCLR." https:// github.com/ sthalles/ SimCLR/ tree/ e8a690ae4f4359528cfba6f270a9226e3733b7fa, 2020.

[24] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," *SIGPLAN Not.*, vol. 44, p. 243–262, Oct. 2009.

[25] N. Grech and Y. Smaragdakis, "P/taint: Unified points-to and taint analysis," in *2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2017, (Vancouver, BC, Canada), pp. 28–35, Association for Computing Machinery, Oct. 2017.

[26] A. Fromherz, A. Ouadjaout, and A. Miné, "Static Value Analysis of Python Programs by Abstract Interpretation," in *10th International Symposium*

NASA Formal Methods, NFM 2018, (Newport News, VA, USA), pp. 185–202, Springer, Apr. 2018.

[27] T. Ball and J. Daniel, "Deconstructing Dynamic Symbolic Execution," in *The 2014 Marktober Summer School on Deop*, no. MSR-TR-2015-95, IOS Press, January 2015.

[28] "huggingface/transformers." https://github.com/huggingface/transformers.

[29] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1* (*Long and Short Papers*), (Minneapolis, Minnesota), pp. 4171–4186, Association for Computational Linguistics, June 2019.

[30] "pfnet-research/ chainer-compiler." `https://github.com/pfnet-research/chainer-compiler/tree/master/tests/elichika_typing/pytorch`. Online: accessed 1 May 2020.

초록

본 논문은 PyTorch 코드에서 텐서 형상 오류를 검출하는 자동 정적분석기 PyTea를 소개한다. 텐서 형상 오류는 한번 일어나면 많은 학습 시간과 중간 결과를 잃어버릴 수 있기에 심층신경망 학습에 있어 매우 중요한 부분을 차지한다. PyTea는 PyTorch 코드를 받아 모든 가능한 실행경로를 정적으로 분석하고, 각 경로마다 텐서 연산이 오류없이 수행될 수 있는 텐서 형상의 조건을 모은 뒤, 그 조건들을 전부 만족시킬수 있는지 없는지를 판단하여 텐서 형상 오류가 있는지를 감지한다. PyTea의 확장성과 정확성은 PyTea의 심볼릭 축약 및 경로 단순화 후 남은 경로 갯수가 많지 않으며, 반복문의 실행 횟수도 충분히 작다는 실제 PyTorch 프로그램의 특성에 기반한다. PyTea는 공식 PyTorch 코드 저장소와 StackOverflow의 텐서 오류 코드를 기반으로 테스트 되었으며, 이러한 실험에서 모두 수 초 이내로 텐서 형상 오류를 검출하였다.

주요어: 정적 분석, 심층 학습, 텐서 형상 오류, SMT 솔버, 파이선, 파이토치

학번: 2020-29856

Acknowledgements

I truly appreciate the guidance of my advisor Chung-Kil Hur, and the professor Kwangkeun Yi. They have assisted me to navigate the vast space of the theories of programming languages and static analysis.

Also, I'd like to thank all the members in Software Foundations Lab and ROPAS Lab, especially the colleagues who have supported this hard works – Sehoon Kim, Woosung Song, Kyuyeon Park, and DongKwon Lee. This dissertation cannot be realized without their great efforts.

This work was partially supported by Korea Institute for Information & Communications Technology Promotion (No.2021-0-00059), NAVER CLOVA (No. 0536-20200005). This work was also supported by BK21 FOUR Intelligence Computing(Dept. of Computer Science and Engineering, SNU) funded by National Research Foundation of Korea(NRF) (4199990214639).