Dynamic Inference of Likely Symbolic Tensor Shapes in Python Machine Learning Programs

Koushik Sen
ksenx@google.com
Google DeepMind
Mountain View, California

Daniel Zheng
danielzheng@google.com
Google DeepMind
Mountain View, California

ABSTRACT

In machine learning programs, it is often tedious to annotate the dimensions of shapes of various tensors that get created during execution. We present a dynamic likely tensor shape inference analysis, called ShapeIt, that annotates the dimensions of shapes of tensor expressions with symbolic dimension values and establishes the symbolic relationships among those dimensions. Such annotations can be used to understand the machine learning code written in popular frameworks, such as PyTorch and JAX, and to find bugs related to tensor shape mismatch. We have implemented ShapeIt on top of a novel dynamic analysis framework for Python, called Pynsy, which works by instrumenting Python bytecode on the fly. Our evaluation of ShapeIt on several tensor programs illustrates that ShapeIt could effectively infer symbolic shapes and their relationships for various neural network programs with low runtime overhead.

KEYWORDS

program analysis, dynamic analysis, program instrumentation, tensor shape inference, dynamic invariant analysis

ACM Reference Format:
Koushik Sen and Daniel Zheng. 2024. Dynamic Inference of Likely Symbolic Tensor Shapes in Python Machine Learning Programs. In *46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3639477.3639718

1 INTRODUCTION

Multi-dimensional arrays, called tensors, are a widely used data structure nowadays. Deep learning and various scientific computing systems have tensors as core data structures. Several popular and widely used deep learning frameworks, such as TensorFlow, PyTorch, JAX, use tensors to represent neural networks. These tensor frameworks provide many valuable and efficient API functions to manipulate and compute on tensors in parallel.

Due to massive AI and deep learning growth, many programmers have started programming using tensors in the last decade. This is a significant switch in the programming paradigm where programmers must use tensors correctly and effectively to program modern machine learning systems. Tensor programming could get complicated and result in bugs not observed in general-purpose programming. A common kind of error that programmers encounter while programming with tensors is runtime tensor shape mismatch error. For example, a tensor reshape operation requires the input and output tensors to have the same number of elements. Thus, if one wants to reshape a tensor of shape $(2, 3)$ to a tensor of shape $(5, 1)$, the operation will throw a runtime shape mismatch error. Such errors arise commonly while writing tensor programs.

Debugging tensor shape mismatch errors is difficult, and a step-by-step debugger might be difficult to use since most tensor frameworks provide lazy evaluation. A common practice to debug such errors is to add print statements to the programs and observe the shape of relevant tensors at runtime. However, such approaches are ad hoc and time-consuming as they require programmers to guess the tensors whose shape they want to print. Another existing approach to avoid such mismatch errors is to perform static shape inference by performing whole program analysis [3, 4, 10, 11]. However, these techniques are too conservative and reject valid programs. Those systems also require annotation of the shapes of the various tensor functions provided by the framework, which could be in hundreds. Moreover, some of these systems could also miss shape mismatch errors. Without robust tools for tensor shape analysis, programmers often resort to manual annotation of shapes of various tensors. Such annotations help with program understanding and debugging any shape mismatch error.

We propose a novel dynamic analysis technique, called ShapeIt, for likely tensor shape inference for tensor programs in Python. The technique infers the relationships among the dimensions of various tensor expressions in a program. The technique is based on the key insight that the dimensions of various tensor expressions in a tensor program can be expressed in terms of a small set of tensor dimensions such as *'batch_size'*, *'height'*, *'width'*, and *'classes'*. For example, consider the tensor expression `y = A * x + b`, where `A`, `b`, `x`, and `y` have the tensor shapes (d_1, d_2), $(d_3,)$, $(d_4,)$, and $(d_5,)$, respectively. For the expression to correctly evaluate, we must have $d_2 = d_3$, $d_1 = d_4$, and $d_1 = d_5$. Therefore, all the dimensions can be expressed in terms of d_1 and d_2. ShapeIt logs the concrete shape of all tensor expressions evaluated during an execution. From the logs, ShapeIt computes the likely relationship among the tensor dimensions and annotates the tensor expressions with their symbolic dimensions expressed in terms of a small set of symbolic tensor dimensions. ShapeIt also allows the user to associate a meaningful name to each dimension in the small set. Then, the annotated code will have all shape annotations expressed in terms of these meaningful names. The annotated code generated

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE-SEIP '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0501-4/24/04.
https://doi.org/10.1145/3639477.3639718

by ShapeIt can be used to understand the shapes of tensor expressions and identify quickly any source of shape mismatch error. The technique works for existing tensor programming libraries such as PyTorch and JAX.

Figure 2 shows the shape-annotated code of a simple tensor program shown in Figure 1. The user has provided meaningful names (i.e., `batch`, `size`, `hidden1`, `hidden2`, `hidden3`, and `classes`) to various dimensions in the first nine lines of code in Figure 1. Those names got propagated to the annotations of other tensor expressions in the program. Note that at line 31 of the annotated code, the shape of `w` changes in each iteration of the loop. Therefore, the shape of `w` cannot be expressed in terms of the meaningful names provided by the user. However, ShapeIt finds that symbolic dimensions of `w` (i.e., `(d2, d3)`) are related to the symbolic dimensions of the tensor expressions `b`, `activations`, `jnp.dot(activation, w)`, and `jnp.dot(activation, w) + b`.

A key challenge in inferring tensor shape relations is that ShapeIt can infer spurious relations because the relations are solely inferred based on the observation of concrete shape values. A simple approach to eliminate false positives would be to run the program with various hyperparameters and use observations from all executions to infer the symbolic shape relations. However, such an approach would be expensive as it requires multiple program executions with different hyperparameters. *We propose a novel approach to avoid such false positives while running the program once.* The approach picks values for hyperparameters dynamically so that no two hyperparameters are equal while not being too far from the original hyperparameter values.

We have implemented ShapeIt on top of Pynsy, a framework that provides useful abstractions and an API that significantly simplifies the implementation of dynamic analyses for Python. The framework is also a novel contribution of the paper and is publicly available at https://github.com/google-research/pynsy. The framework works through Python bytecode instrumentation and allows the implementation of various heavy-weight dynamic analysis techniques. Pynsy can be used to develop various dynamic analyses for Python similar to AddressSanitizer[21], Valgrind[18], Pin[16] for C/C++ programs. Such dynamic analyses could help us annotate code or execution with useful program properties under analysis.

There are a few advantages of using dynamic instrumentation of Python bytecode.

- The analysis works for any Python code, whether its source is available. Moreover, since the instrumentation of the Python bytecode happens on the fly, there is no need to write compiler passes required to instrument source code. The analysis could be easily used and deployed since we don't need the extra instrumentation phase to process the Python source code.
- Unlike Python source code, Python bytecode is small and does not change from version to version, mostly. Therefore, it is easy to maintain a dynamic analysis framework that operates on Python bytecode. One could argue that we could modify the Python interpreter to collect necessary runtime information. However, the Python interpreter evolves rapidly, making maintaining such a code base difficult. Moreover, people are also reluctant to use a modified Python interpreter for safety and security reasons.
- Pynsy's framework independent instrumentation enables us to implement ShapeIt once and for all for various tensor frameworks such as PyTorch and JAX. One could use these frameworks, which already support tracing, to implement ShapeIt. However, such implementations would be different for each framework. Moreover, they will be difficult to maintain as the frameworks evolve.

We have evaluated Pynsy on several deep-learning programs written in JAX and PyTorch. Our results show that Pynsy has an average overhead of up to 5×. In a manual case study with three programs already annotated with shape information by their programmer, we found that Pynsy derived shape annotations match human annotations already present in the program. Finally, we show that Pynsy helps to reduce the number of symbolic shape dimensions in our benchmark programs significantly.

2 SHAPEIT: DYNAMIC LIKELY TENSOR SHAPE INFERENCE ALGORITHM

ShapeIt works in several phases described in the next subsections.

2.1 Dynamic data collection

We assume that a program has a set of functions and methods, say F, and each function $f \in F$ has a set of expressions, each of which evaluates to a tensor object. Such expressions are called tensor expressions. The expressions are identified by their static *location* in the program. Let L be the set of locations of all tensor expressions in the program. Since ShapeIt is a dynamic analysis algorithm, it can determine if an expression evaluates to a tensor object at runtime. For example, the function `init_random_params` in Figure 1 has two expressions that evaluate to tensor objects: `rng.randn(m,n)` and `rng.randn(n)`. Each tensor object has a shape which is a tuple of the form $(n_1, n_2, \ldots, n_r)$ where each $n_i \in \mathbb{N}$. The length of the tuple (i.e., r) gives the tensor object's rank (i.e., the number of axes), and the tuple's elements give the dimension of each axis.

ShapeIt instruments the program under analysis so that when the instrumented program is executed, the instrumentation logs the concrete shape of each tensor resulting from the execution of each tensor expression. A *trace* is a map from L (i.e., the set of all locations of tensor expressions) to a concrete shape value. For example, a partial trace of the execution of the program in Figure 1 is shown in Figure 3. The trace is restricted to the execution of the function `init_random_params()` for simplicity. The trace records the shape values of the tensor expressions `rng.randn(m,n)` and `rng.randn(n)` at line 22. The locations of these tensor expressions are, say, 3 and 4, respectively. Since each of the two expressions gets evaluated three times in a loop, the trace records six shape values of the tensor expressions.

2.2 Creating symbolic dimension variables

For each tensor expression, if the concrete shape value of the expression has been observed to have rank r, then ShapeIt introduces fresh symbolic dimension variables for each dimension of

```
1  inputs = rngi.randn(128, 784)
2  layer_sizes = [784, 1024, 1024, 10]
3
4  def init_random_params(layer_sizes, rng=npr.RandomState(0)):
5    return [
6        (rng.randn(m, n), rng.randn(n))
7      for m, n, in zip(layer_sizes[:-1], layer_sizes[1:])
8    ]
9
10 def predict(params, inputs):
11   activations = inputs
12   for w, b in params[:-1]:
13     outputs = jnp.dot(activations, w) + b
14     activations = jnp.tanh(outputs)
15
16   final_w, final_b = params[-1]
17   logits = jnp.dot(activations, final_w) + final_b
18   return logits - logsumexp(logits, axis=0, keepdims=True)
19
20 predict(init_random_params(layer_sizes), inputs)
```

Figure 1: A simple tensor program extracted from a JAX implementation of MNIST.

the tensor. For example, if the concrete shape of a tensor expression at line 19 is recorded as $(10, 28, 28)$, then ShapeIt introduces the symbolic shape (d_1, d_2, d_3) where d_1, d_2, and d_3 are fresh symbolic dimension variables. The symbolic shape of a tensor expression remains the same throughout an execution, although the concrete shape of the expression could be different at different points in the execution. The goal of ShapeIt is to establish the likely symbolic relationships between these symbolic dimension variables based on the concrete shape values observed during an execution. Let D be the set of all fresh symbolic dimension variables introduced during the execution of a program. For example, ShapeIt introduces the fresh symbolic dimension variables d_0, d_1, and d_2 to denote the dimensions of the tensor expressions `rng.randn(m,n)` and `rng.randn(n)` at line 22.

Note that we are creating symbolic dimension variables to denote the shape of each tensor expression and not the shape of each program variable. A tensor expression can have different ranks at different points of execution. We can detect this case by looking at all the shape values observed for the tensor expression in an execution. If a tensor expression has different shape ranks, then ShapeIt cannot work with such tensor expressions. We do not create any symbolic dimension for that tensor expression in such cases. In practice, this does not happen. On the other hand, a program variable can have different ranks at different execution points in real-world programs. Therefore, we are not inferring shapes for program variables.

2.3 Abstract state creation

The goal of abstract state creation is to create an abstract state at each point in the trace. Such an abstract state maps a symbolic dimension variable to its latest observed concrete value. ShapeIt checks which relationships among the dimension variables are satisfied by all the abstract states in the trace and reports them as the final result of the analysis.

The abstract state of a program, say Σ, at any point in the trace is the latest shape value of the live tensor expressions during the

```
1  # ▷ randn: [batch, size]
2  inputs = rngi.randn(128, 784)
3  # ▷ inputs: [batch, size]
4  # ▷ annotate_shape: [batch, size]
5  shaper.annotate_shape(inputs, ("batch", "size"))
6
7  layer_sizes = [
8      # ▷ hyper_parameter: [size]
9      shaper.hyper_parameter(784, "hidden1"),
10     # ▷ hyper_parameter: [hidden2]
11     shaper.hyper_parameter(1024, "hidden2"),
12     # ▷ hyper_parameter: [hidden3]
13     shaper.hyper_parameter(1024, "hidden3"),
14     # ▷ hyper_parameter: [classes]
15     shaper.hyper_parameter(10, "classes"),
16 ]
17
18 def init_random_params(layer_sizes, rng=npr.RandomState(0)):
19   return [
20       # ▷ randn: [d0, d1]
21       # ▷ randn: [d1]
22       (rng.randn(m, n), rng.randn(n))
23     for m, n, in zip(layer_sizes[:-1], layer_sizes[1:])
24   ]
25
26 def predict(params, inputs):
27   # ▷ inputs: [batch, size]
28   activations = inputs
29   for w, b in params[:-1]:
30     # ▷ activations: [batch, d2]
31     # ▷ w: [d2, d3]
32     # ▷ dot: [batch, d3]
33     # ▷ b: [d3]
34     # ▷ +: [batch, d3]
35     outputs = jnp.dot(activations, w) + b
36     # ▷ outputs: [batch, d3]
37     # ▷ tanh: [batch, d3]
38     activations = jnp.tanh(outputs)
39
40   final_w, final_b = params[-1]
41   # ▷ activations: [batch, hidden3]
42   # ▷ final_w: [hidden3, classes]
43   # ▷ dot: [batch, classes]
44   # ▷ final_b: [classes]
45   # ▷ +: [batch, classes]
46   logits = jnp.dot(activations, final_w) + final_b
47   # ▷ logits: [batch, classes]
48   # ▷ logsumexp: [batch, classes]
49   # ▷ -: [batch, classes]
50   # ▷ return: [batch, classes]
51   return logits - logsumexp(logits, axis=0, keepdims=True)
52
53 # ▷ inputs: [batch, size]
54 # ▷ predict: [batch, classes]
55 predict(init_random_params(layer_sizes), inputs)
```

Figure 2: An annotated version of the tensor program in Figure 1 with shape annotations inferred by ShapeIt. ShapeIt-inferred shape annotations are shown in comment lines with ▷. Additional shaper operations are added to annotate known dimension names (to improve the readability of inferred annotations) and hyperparameter dimensions (to avoid false positive constraints, see Section 2.6).

execution up to the point in the trace. A trace maps D, the set of all fresh symbolic dimension variables, to their concrete dimension values observed during the execution. When a function f is called, the current abstract state is copied, and all symbolic dimension variables corresponding to tensor expressions inside the function are reinitialized to $\perp$ (i.e., undefined). The newly created abstract

```
line 22: trace(3) = (784, 1024)
line 22: trace(4) = (1024,)
line 22: trace(3) = (1024, 1024)
line 22: trace(4) = (1024,)
line 22: trace(3) = (1024, 10)
line 22: trace(4) = (10,)
```

Figure 3: A partial trace of the program in Figure 1.

```
2: {d0: ⊥, d1: ⊥, d2: ⊥}
3: {d0: 784, d1: 1024, d2: ⊥}
4: {d0: 1024, d1: 1024, d2: 1024}
3: {d0: 1024, d1: 10, d2: 1024}
4: {d0: 784, d1: 1024, d2: 1024}
3: {d0: 1024, d1: 1024, d2: 1024}
4: {d0: 1024, d1: 10, d2: 10}
```

Figure 4: Abstract states after the evaluation of each tensor expression in Figure 1 at line 22.

state becomes the current state of the execution. The abstract state is also pushed to a call stack. When a tensor expression in the function is evaluated during the execution, the corresponding symbolic dimension variables in the current abstract state are updated to the observed dimension values. When the function returns, the current abstract state is popped from the call stack, and the state of the caller function, which is at the top of the stack, becomes the current abstract state of the program.

For example, let us assume that the symbolic dimension variables associated with the tensor expressions at line 22 are (d_0, d_1) for `rng.randn(m,n)` and $(d_2,)$ for `rng.randn(n)`. Then, the abstract state just before the execution of the partial trace in Figure 3 is given by the first line in Figure 4. The second line gives the abstract state after the evaluation of the tensor expression `rng.randn(m,n)` and the third line gives the state after the evaluation of the tensor expression `rng.randn(n)`. Each line starts with a number. The number is the location of the tensor expression whose last evaluation resulted in the state.

2.4 Anti-unification

After collecting a trace, creating symbolic dimension variables, and constructing the abstract states along the collected trace, ShapeIt checks if certain symbolic relationships consistently hold among the symbolic dimension variables at a program location in all abstract states. ShapeIt has a pre-defined set of templates of symbolic relations, which are as follows:

- $d_i = d_j$, a template stating that symbolic dimensions d_i and d_j have the same concrete value in all abstract states if neither is $\perp$,
- $d_i = d_j \cdot d_k$, a template where the symbolic dimension d_i is the product of the dimensions d_j and d_k if neither is $\perp$,
- $d_i = d_j + d_k$, a template where the symbolic dimension d_1 is the sum of the dimensions d_j and d_k if neither is $\perp$,
- $d_i = d_j \cdot d_k / d_l$, a template where $d_i \cdot d_l = d_j * d_k$.

More templates can be added by the user of ShapeIt; however, in practice, we found these templates to be sufficient to infer the common relationships among the symbolic dimension variables.

Given the set of templates, ShapeIt's anti-unification computes all the relationships described by the templates that hold throughout the execution. Specifically, ShapeIt runs the following in a loop:

(1) For each template $t_{i_1 \ldots i_m}$, where $i_1 \ldots i_m$ are the symbolic dimensions involved in the template,
(2) For each location ℓ of a tensor expression,
(3) For each $d_1, \ldots, d_m \in D$, where at least a d_i is a symbolic dimension at the location ℓ,
(4) For all abstract states Σ at location ℓ, (i.e., the abstract state after the evaluation of the tensor expression at ℓ), if $\Sigma(d_1), \ldots, \Sigma(d_m)$ makes the template $t_{i_1 \ldots i_m}$ true, then the template instantiated with the symbolic dimensions $d_1, \ldots, d_m$ is a likely relationship computed among the symbolic dimensions.

The set of all likely constraints returned by ShapeIt are likely symbolic relationships satisfied by the tensor objects in the program execution. ShapeIt augments the code with the information about shape relationships after each program location.

For example, consider the template $d_i = d_j$ with $i = 7$ and $j = 8$ and location 4. Then, from the sequence of states in Figure 4, we can see that $d_1 = d_2$ at location 4 for all three states at location 4. Therefore, we can say that $d_1 = d_2$ is a likely relationship inferred from the execution of the program. On the other hand, consider the same template with $i = 6$ and $j = 8$. Then, at location 3, only two of the three states satisfy the relationship. Therefore, $d_0 = d_2$ is not a valid relationship.

2.5 Substitution and named dimensions

ShapeIt allows its users to annotate shape dimensions with meaningful names in two ways: using `shaper.annotate_shape(t, (name1, name2, ...)` which specifies that the dimensions of the tensor `t` has the names `name1, name2, ...`, respectively, and using `shaper.hyper_parameter(dim, name)` which specifies that the dimension value passed as the first argument has the name given by the second argument. The second API call returns the first argument.

Once ShapeIt has computed all the symbolic relationships among the symbolic dimensions, it tries to reduce the number of dimension variables via substitutions. Let C be the set of constraints on dimension variables found by ShapeIt. The substitution algorithm works as follows.

(1) If a dimension d is annotated with a name, say `name`, then any occurrence of d in C is replaced with `name`.
(2) If $d_i = d_j$ is a constraint in C, then replace any occurrence of d_i with d_j if $i > j$.
(3) If $d_i = e$ is a constraint where e is an expression involving two or more dimension variables, then replace any occurrence of d_i with e in C.
(4) The previous two rules are applied repeatedly on C until no more substitutions can be performed in C. The first rule breaks the symmetry and ensures that the loop terminates.

ShapeIt then annotates the shape of every tensor expression in the program with the reduced set of dimension variables.

For example, for the trace in Figure 3 ShapeIt will compute the relationship $d_2 = d_1$. Therefore, it will annotate the tensor expressions `rng.randn(m,n)` and `rng.randn(n)` with (d_0, d_1) and $(d_1,)$, respectively, showing that the dimension of the 1^{st} axis of `rng.randn(m,n)` is same as the dimension of the 0^{th} axis of `rng.randn(n)`.

2.6 False positive avoidance

In ShapeIt, since we are generating likely constraints based on observed values, it is possible to infer wrong relations if, by chance, the observed values support the relation, but the relation may not hold if the hyperparameters of the program are changed. For example, consider the case where all the `layer_sizes` in Figure 1 are equal to 784. In such a case, ShapeIt will infer that the tensor expressions `rng.randn(m,n)` and `rng.randn(n)` at line 22 have symbolic shapes `(size, size)` and `(size,)` which is true for the current execution, but may not be true if any of the layer sizes is not 784. We, therefore, report a false positive for the shapes of these tensor expressions.

One naive way to eliminate such false positives is to run the program multiple times with different sets of hyperparameters and infer the symbolic shapes by combining the observations from all executions. However, this approach is inefficient as it requires executing the program several times with different hyperparameters. In ShapeIt, we propose a novel approach to avoid such false positives. The approach is based on the insight that if a tensor program is run with different hyperparameters, the execution remains semantically correct, although the actual accuracy of training a neural network may change. We assume the user has already annotated all the dimensions representing hyperparameters with meaningful names using the `shaper.hyper_parameter` function. During the execution, whenever `shaper.hyper_parameter(dimension_value, dimension_name)` is executed, ShapeIt observes if the `dimension_value` returned by other previous calls to `shaper.hyper_parameter` is same as the `dimension_value` in the current call. If this is the case, ShapeIt returns the minimum value that is greater than `dimension_value` and is not equal to the `dimension_value` returned by any previous call to `shaper.hyper_parameter`. This ensures that all hyperparameters are distinct while not being too different from the actual hyperparameters. This, in turn, ensures that the relations that are inferred due to the same value of some hyperparameters are no longer inferred. A key advantage of dynamically choosing different hyperparameters in a single execution is avoiding false positives even if the observations are made using a single execution.

3 PYNSY: IMPLEMENTATION

We have implemented ShapeIt on top of a dynamic analysis framework for Python called Pynsy. In addition to ShapeIt, Pynsy is also a novel contribution of the paper. Pynsy instruments the Python bytecode of a target application on the fly and provides a hook to log or inspect each Python bytecode instruction being executed, along with dynamic information about the operands involved in the instruction. A custom dynamic analyzer can be implemented by overriding the hooks provided by Pynsy. ShapeIt has been implemented as a dynamic analyzer in Pynsy. We have made Pynsy and ShapeIt implementations open-source at https://github.com/google-research/pynsy.

```python
def abstraction(obj: Any) -> tuple[bool, Any]:
  """Returns an abstract representation of the given object.

  Args:
    obj: The object to abstract.

  Returns:
    A tuple (bool, Any) where the first value indicates whether
    the abstraction should track the location of the object, and
    the second value is a finite abstraction of the object.
  """

def process_event(record):
  """Process each instrumentation event as it is generated."""

def process_termination():
  """Called at the end of an analysis."""
```

Figure 5: The interface for writing a custom dynamic analyzer for Python in Pynsy.

3.1 Pynsy API for Dynamic Analysis

One can write a custom dynamic analysis for Python programs by creating a Pynsy analysis module, which defines the functions shown in Figure 5. The function `abstraction` takes any Python object and should return an abstraction of the object's value that the custom analysis is interested in. For example, in the case of ShapeIt, the abstraction should return the shape tuple if the object has a shape attribute. Note that this simple abstraction function, which does not require the object to be a tensor type, makes it compatible with PyTorch and JAX. The custom abstraction is returned by the `abstraction` function as the second component of the returned pair. The first component of the return pair hints at whether to include the unique object ID of the object in the abstraction. We never used the object ID of an object in ShapeIt, so we kept the first returned tuple component true. Note that the custom abstraction function would be different for different dynamic analyses. For example, if we want to perform a program's likely dynamic type inference, the abstraction functions should return the object type.

When a Pynsy-instrumented program is executed, it records information about every load, store, and application (e.g., application of a binary/unary operator or invocation of a method) instruction executed by the program in order. For each bytecode instruction being executed by the Python interpreter, Pynsy calls the hook function `process_event(record)`. A custom dynamic analysis would override this function to implement the analysis. The record structure passed as an argument to `process_event(record)` contains static and dynamic information about the executed bytecode instruction. Each record has the following attributes:

- `module_name`: The name of the module whose bytecode instruction execution generated the record.
- `method_id`: The unique method id whose instruction has generated the record.

```python
def abstraction(obj):
  if (
      isinstance(obj, tuple)
      or isinstance(obj, list)
      or str(type(obj)) == "<class 'range'>"
  ):
    return True, len(obj)
  else:
    return True, None


def process_event(record):
  if record.opcode == "CONTAINS_OP":
    list_len: int = record.results_and_args[3].abstraction
    module_name = record.module_name
    lineno = record.lineno
    if list_len is not None and list_len > 100:
      print(
          "Warning: key in list is slow for a list of length "
          f"{list_len} at {module_name}:line {lineno}"
      )
```

Figure 6: A "key in list" expensive check dynamic analysis in Pynsy.

- `instruction_id`: The unique instruction within the method that generated the record.
- `lineno`: The line number of the program such that compilation of the statement at the line number resulted in the instruction bytecode.
- `type`: Type of the bytecode instruction such as `LOAD_FAST`, `STORE_FAST`, `CALL_FUNCTION`.
- `indentation`: The indentation of the instruction being executed. This helps to capture the recursive organization of the instructions.
- `before`: Whether the record appears before executing the instruction or not.
- `result_and_args`: A list containing abstractions of the result produced by the execution of the instruction and the arguments being used by the instruction.
- `name`: The name of the variable or attribute if the instruction accesses the value of a variable or attribute.
- `function_name`: The function's name being called.

Once Pynsy has executed the entire program, it calls the hook function `process_termination`. A custom dynamic analysis for Python could override this function to perform the final analysis and produce reports.

3.2 A Pynsy dynamic analysis example

Pynsy supports defining custom dynamic analyses. We illustrate how to implement the KeyInList analysis from DynaPyt [6]. In Python, checking whether a key exists in a list is expensive and takes linear time compared to the sub-linear time for a set or dict. Therefore, an analysis that finds all program locations where a key in a long list is queried would be useful to eliminate inefficiencies in the program.

Figure 6 shows a dynamic analysis for detecting expensive "key in list" operations. The analysis only cares about objects of type list, tuple, or range. For these objects, the `abstraction` function returns the object's length. Otherwise, `abstraction` returns `None`.

The function `process_event` performs the actual check. If the operator is `in`, whose bytecode is `CONTAINS_OP`, and if the abstraction of the second operand of the bytecode is not None and greater than 100 (where lists with 100 or more elements are considered long lists), we report a warning to the user. Note that this analysis cannot be implemented precisely using static analysis because Python is dynamically typed, and one cannot always precisely infer if the type of the second operand for the `in` operator is a list.

4 EVALUATION

We run experiments with ShapeIt implemented in Pynsy to answer the following questions:

- **RQ1** Runtime overhead: what is the runtime overhead of instrumenting programs and running tensor shape inference?
- **RQ2** Shape inference correctness: are inferred shapes consistent with human-written shape annotations?
- **RQ3** Shape inference statistics: how many unique dimension variables remain after anti-unification, compared with the number of total starting dimensions?

We evaluate ShapeIt on a suite of publicly available programs using popular machine learning libraries, including PyTorch and JAX (Flax [13], Haiku [15], Haliax[1], and Levanter[2]).

The experiments were performed on a virtual machine with 2.2 GHz Intel Xeon and 16 GB RAM running Debian. Python 3.10 was used for all experiments.

4.1 Runtime overhead (RQ1)

We evaluate the runtime performance of ShapeIt by comparing the original runtime of a suite of machine learning programs versus the runtime with ShapeIt instrumentation and analysis.

Since ShapeIt involves program instrumentation via Pynsy, some performance overhead is expected. Table 1 shows the relative cost of ShapeIt instrumentation: slowdown is between 1× and 5×, which is on par with similar program analysis frameworks. For comparison, the TraceAll analysis from Dynapyt [6] is reported to have a slowdown between 1.2× and 16×. Compared with frameworks for other languages, the Jalangi framework for JavaScript [20] imposes 26×-30× overhead, and the RoadRunner framework for Java bytecode [8] imposes an average instrumentation overhead of 52×.

As a software development tool for analyzing shapes, ShapeIt is intended to be run incrementally to infer shapes for new or modified code units, not repeatedly during end-to-end executions of machine learning programs, so slowdown is not a critical concern for most use cases. Since ShapeIt operates on program traces, the algorithm is agnostic to how traces are collected and could be implemented with an instrumentation approach that collects lower-quality traces with less overhead.

[1] https://github.com/stanford-crfm/haliax
[2] https://github.com/stanford-crfm/levanter
[2] https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt

ML framework	Name	Baseline program	With Pynsy and ShapeIt	Slowdown
JAX	flax/mnist	62.65	71.64	1.14×
	haiku/impala_rl	58.23	64.61	1.11×
	haiku/rnn	13.93	17.80	1.28×
	haiku/transformer	80.84	86.26	1.07×
	levanter/gpt2	61.80	137.05	2.21×
PyTorch	pytorch/gcn	9.94	27.56	2.77×
	pytorch/mnist_forward_forward	24.98	53.91	2.16×
	pytorch/regression	2.44	10.93	4.47×
	pytorch/siamese_network	11.32	34.30	3.03×
	pytorch/vae	13.66	45.11	3.30×
	pytorch/vision_transformer	13.80	14.60	1.06×

Table 1: End-to-end CPU runtime performance on benchmark programs (in seconds) with and without ShapeIt instrumentation. The slowdown is between 1x and 5x when instrumenting ML programs for one execution of one training loop iteration, which is a realistic usage mode for ShapeIt. All numbers are averaged across three runs.

4.2 Shape inference correctness (RQ2)

Next, we look at the ability of ShapeIt to infer shapes that are correct. We measure the correctness of inferred shapes by comparing them against human-written shape annotations on a suite of programs.

Given an inferred shape s_i and a target human-annotated shape s_t, we propose an "equivalent match" criterion, which is satisfied if s_i and s_t have dimension variables that are (1) identical or (2) equal in value, in the case where dimension variables have different explicitly-annotated names but the same concrete values at all abstract states.

For example, in Figure 1: the `size` and `hidden1` dimensions have different names but the same value of 784, so we treat s_i = (batch, size) and s_t = (batch, hidden1) as equivalent when measuring inferred shape correctness.

In practice, not all machine learning programs use shape annotations, including most of those evaluated in Table 1. (We believe this is partly due to a lack of easy-to-use shape inference tools, given the value of shape annotations as documentation.) Most human-written shape annotations take the form of informal code comments, which serve as documentation and are unchecked.

To expand the set of programs used for evaluation, we manually annotate some programs from Section 4.1 following standard conventions for writing shape annotations. This involves annotating shapes of function parameters and results and shapes of top-level program inputs.

Table 2 shows correctness metric results for ShapeIt. We find that ShapeIt is generally able to infer precise shape annotations in practice, consistent with handwritten shape annotations by programmers in real machine learning programs. In `flax/wmt` and `levanter/gpt2`, ShapeIt does not achieve 100% equivalent match for shape annotations due to false positive constraints being inferred between dimension variables with small constant values like 1 and 2: this leads to annotations with dimension expressions that are less precise, but improving these is possible with heuristics in anti-unification.

4.3 Shape inference statistics (RQ3)

Finally, we examine quantitative metrics from running ShapeIt on real-world programs to gain insight into the algorithm's behavior in practice.

We measure the following metrics for ShapeIt related to anti-unification:

- The total number of starting dimension variables before anti-unification.
- The number of dimension variables after anti-unification.
- The number of unique dimension expressions after anti-unification.
- The number of unique dimension names in human-written shape annotations—if human annotations are available.

Table 3 shows these metrics for a suite of programs. We see that ShapeIt is able to anti-unify a large fraction of dimension variables in real-world programs. In practice, ShapeIt infers shapes for all intermediate program values and often introduces many dimension variables—more than that are helpful for human programmers—so shapes containing dimension variables that are not anti-unified or not explicitly named can simply be discarded instead of being shown to programmers.

One observation from our study is that careful annotation is important: poor shape annotations, like annotating constant dimensions, can cause spurious false positive constraints to be generated, lowering the overall quality of inferred shapes. For example, pseudorandom number generation key (PRNGKey) values are internally represented in JAX as tensors with shape $(2,)$. While it is possible to explicitly shape-annotate PRNGKey values with `shaper.annotate_shape(key, ("key",))`, this causes unhelpful d_i = key $\cdot$ d_j constraints to be generated, for even-valued dimension variables d_i. Instead, writing annotations of API significance (e.g., on function parameters and results) avoids these issues and leads to practically helpful inferred shapes.

Name	LOC	Human-annotated shapes	ShapeIt-inferred annotations	% Equivalent match
jax/mnist	69	12	80	12/12 = 1.0
flax/wmt	262	42	182	38/42 = 0.90
levanter/gpt2	494	45	185	41/45 = 0.91

Table 2: Evaluation of ShapeIt correctness vs human-annotated programs. ShapeIt infers shape annotations for all intermediate tensor values in programs, while humans typically annotate only a subset. "Equivalent match" means "exact value match" in the case where dimension variables have different explicit names but the same concrete value in all abstract states. Lines-of-code (LOC) are measured using ohcount [2].

Name	Annotations	Total starting dims.	Unique dim. expressions	Unique dims.	Human-named dims.
flax/mnist	90	172	34	32	...
haiku/impala_rl	85	96	31	28	...
haiku/rnn	18	33	3	3	...
haiku/transformer	96	220	7	6	...
levanter/gpt2	185	405	19	16	10
pytorch/gcn	147	208	17	17	...
pytorch/mnist_forward_forward	116	173	22	22	...
pytorch/regression	46	64	4	4	...
pytorch/siamese_network	104	206	17	17	...
pytorch/vae	89	196	34	33	...
pytorch/vision_transformer	81	231	27	25	...

Table 3: Anti-unification metrics for ShapeIt. Given a large number of initial starting dimension variables, ShapeIt is capable of inferring dimension relationships and identifying a small set of unique dimension variables. Some programs involve dimension expressions like $d_j \cdot k$ and $d_j + k$.

5 RELATED WORK

Program invariant detection. There is a large body of work on automatic inference of program invariants, including both static [9] and dynamic approaches [5, 7]. ShapeIt is similar to the dynamic approaches. Daikon [7] infer invariants over the program variables. Therefore, these techniques can introspect the state of the program dynamically and check various invariant templates. ShapeIt infers invariants over shape dimension variables. Such variables are not program variables but are variables introduced by ShapeIt. Therefore, ShapeIt needs to create the state of the shape variables by observing a trace. Moreover, the set of templates used by ShapeIt is much smaller and simpler than in Daikon.

Python instrumentation and dynamic analysis. The Python standard library offers a `sys.settrace` function [3] for registering a hook that gets called at one of three granularity levels: at every executed opcode, line of code, or function call. However, for bytecode-level tracing, `sys.settrace` provides only opcodes without any information about the value of the operands and result, making it insufficient for producing fine-grained object-level program traces. Dynapyt [6] is a general-purpose dynamic analysis for Python that explicitly chooses to do source-level instrumentation via AST rewriting to express analyses at a high level of abstraction on a stable code representation. Our Pynsy framework does bytecode-level instrumentation. Therefore, Pynsy works for any Python code, whether its source is available. Moreover, since the instrumentation of the Python bytecode happens on the fly, there is no need to write compiler passes required to instrument source code. Unlike Python source code, Python bytecode is small and does not change from version to version mostly. Therefore, it is easy to maintain a dynamic analysis framework that operates on Python bytecode.

Tensor shape annotation and checking. . There has been work on type systems for tensor shape safety, including gradual typing [11], refinement types [10], and named tensor DSLs [3, 4]: these systems focus on shape checking in statically-typed languages. Gradual tensor shape typing [11] is particularly related to our work, as it combines best-effort static inference with dynamic checks; ShapeIt could fit into a gradual typing system by incrementally suggesting annotations to improve inference precision.

Dynamic shape fault detection. ShapeFlow [22] is a fork of TensorFlow that uses dynamic abstract interpretation to detect shape faults. In a TensorFlow program, one could replace TensorFlow with ShapeFlow. ShapeFlow will then not compute the actual tensors but the shape of the tensors. ShapeFlow's analysis is precise; however, it requires one to write an abstract interpreter for the computation graph, which requires a TensorFlow-specific implementation. Moreover, ShapeFlow requires one to modify the code of 118 TensorFlow APIs so that those APIs do only shape computation. Porting ShapeFlow to other frameworks would be, therefore,

[3] https://docs.python.org/3/library/sys.html#sys.settrace

tedious. ShapeIt does not require any modification to the APIs because it tries to infer the symbolic shape dimensions of the API calls based on runtime observation. Elichika [12] uses a similar method to ShapeFlow but applied to PyTorch, with a feature to display the interpreted shapes with a symbolic expression. However, like ShapeFlow, Elichika needs the specification of the API methods to perform symbolic shape inference. ShapeIt has no such limitation as it tries to infer such likely specifications by observing the concrete values only at the call sites of the API. ShapeIt can also be used for shape fault detection, general across machine learning frameworks, and including for buggy programs. This would work by collecting program traces up to an exception, then displaying inferred shape annotations like in Figure 2 to let users identify unexpected dimensions and work backward to find where they appear.

Python type inference. Traditional type inference approaches are rule-based and rely on statically resolved types for accurate inference; these do not transfer well to dynamic languages like Python, where the types of many variables cannot be resolved statically. Several recent projects [1, 14, 17, 19] explore general-purpose type inference in Python and other dynamic languages, primarily via machine-learning-based approaches.

In Python, type annotations have been supported as a language feature since version 3.5, and many libraries have been developed for annotating types of multi-dimensional arrays using the `typing` module and variadic generics. Modern libraries like jaxtyping[4] support precise dimension-level tensor shape annotations and are compatible with static and runtime type checkers. However, these approaches only enable tensor shape *checking* and do not provide shape *inference* beyond straightforward type propagation between variable assignments and shape-preserving functions. To our knowledge, our work is the first to explore tensor shape inference from dynamic traces. Rather than being at odds, ShapeIt can interact nicely with existing tensor shape annotation libraries: future development can extend ShapeIt to produce annotations in a library-supported format to help developers convert unannotated or partially annotated programs to fully annotated programs by adding one shape annotation at a time.

6 CONCLUSION

We present ShapeIt, a novel algorithm for symbolic tensor shape inference in machine learning programs. We also develop Pynsy, a library for heavyweight bytecode-level dynamic analysis in Python, and use it to implement ShapeIt. We show that ShapeIt in Pynsy can infer precise and useful symbolic tensor shape annotations for real-world machine learning programs without manual annotations. In future work, we plan to continue developing ShapeIt into a polished and practical shape linter tool for machine learning practitioners to extend the anti-unification algorithm to support named variadic dimensions (such as `batch...`) for rank-polymorphic annotations, and to conduct a case study with users on applying the tool to large-scale real-world codebases.

[4]https://github.com/google/jaxtyping

7 ACKNOWLEDGEMENTS

We thank Miltos Allamanis and Charles Sutton for their insightful comments and suggestions.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. https://doi.org/10.1145/3385412.3385997

[2] Inc. Black Duck Software. 2019. *Ohcount: Ohloh's source code line counter.* https://github.com/blackducksoftware/ohcount

[3] Tongfei Chen. 2017. Typesafe Abstractions for Tensor Operations (Short Paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala (SCALA 2017)*. 45–50. https://doi.org/10.1145/3136000.3136001

[4] Breandan Considine, Michalis Famelis, and Liam Paull. 2019. *KotlinΔ: A Shape-Safe eDSL for Differentiable Programming.* https://doi.org/10.5281/zenodo.3549076

[5] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic Symbolic Execution for Invariant Inference. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 281–290. https://doi.org/10.1145/1368088.1368127

[6] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: A Dynamic Analysis Framework for Python. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 760–771. https://doi.org/10.1145/3540250.3549126

[7] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.

[8] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toronto, Ontario, Canada) *(PASTE '10)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1806672.1806674

[9] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME '01)*. Springer-Verlag, Berlin, Heidelberg, 500–517.

[10] Yanjie Gao, Zhengxian Li, Haoxiang Lin, Hongyu Zhang, Ming Wu, and Mao Yang. 2022. Refty: Refinement Types for Valid Deep Learning Models. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1843–1855. https://doi.org/10.1145/3510003.3510077

[11] Momoko Hattori, Naoki Kobayashi, and Ryosuke Sato. 2023. Gradual Tensor Shape Checking. arXiv:2203.08402 [cs.PL]

[12] Momoko Hattori, Shimpei Sawada, Shinichiro Hamaji, Masahiro Sakai, and Shunsuke Shimizu. 2020. 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@PLDI 2020, London, UK, June 15, 2020*, Koushik Sen and Mayur Naik (Eds.). ACM, 11–19. https://doi.org/10.1145/3394450.3397465

[13] Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. 2023. *Flax: A neural network library and ecosystem for JAX.* http://github.com/google/flax

[14] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/3236024.3236051

[15] Tom Hennigan, Trevor Cai, Tamara Norman, Lena Martens, and Igor Babuschkin. 2020. *Haiku: Sonnet for JAX.* http://github.com/deepmind/dm-haiku

[16] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[17] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py. In *Proceedings of the 44th International Conference on Software Engineering*. ACM. https://doi.org/10.1145/3510003.3510124

[18] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[19] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. 2023. Generative Type Inference for Python. arXiv:2307.09163 [cs.SE]

[20] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[21] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[22] Sahil Verma and Zhendong Su. 2020. ShapeFlow: Dynamic Shape Interpreter for TensorFlow. *CoRR* abs/2011.13452 (2020). arXiv:2011.13452 https://arxiv.org/abs/2011.13452