Dimspector: Static Shape Inference in PyTorch

https://github.com/nwrousell/dimspector

NOAH ROUSELL, Brown University, Rhode Island
PRACCHO MUNA-MCQUAY, Brown University, Rhode Island

Why Shape Inference?

- **Tensor shapes are difficult to reason about**, especially with broadcasting and reshaping.
- Discovering a shape error *after hours of training* is costly.
- A developer tool should make this reasoning explicit and detect bugs early.
  → `jaxtyping` checks shapes dynamically at function boundaries.
  → `PyTea` performs static analysis, but relies on extensive hand-written models and an SMT solver.

Spot the Bug

Users annotate parameters with [shapes]:

```
1 def relative_attention_cross(
2     x_q: [batch, seq_q, d_model], x_kv: [batch, seq_k, d_model], W_q: [d_model, heads, d_k],
3     W_k: [d_model, heads, d_k], W_v: [d_model, heads, d_v], rel_pos_bias: [seq_k, seq_q]
4 ):
5     query: [batch, heads, seq_q, d_k] = torch.transpose(torch.reshape(x_q @ torch.reshape(W_q, (x_q.shape[2],
        -1)), (x_q.shape[0], x_q.shape[1], W_q.shape[1], W_q.shape[2])), 1, 2)
6     key: [batch, heads, seq_k, d_k] = torch.transpose(torch.reshape(x_kv @ torch.reshape(W_k, (x_kv.shape[2],
        -1)), (x_kv.shape[0], x_kv.shape[1], W_k.shape[1], W_k.shape[2])), 1, 2)
7     value: [batch, heads, seq_k, d_v] = torch.transpose(torch.reshape(x_kv @ torch.reshape(W_v, (x_kv.shape[2],
        -1)), (x_kv.shape[0], x_kv.shape[1], W_v.shape[1], W_v.shape[2])), 1, 2)
8     scores: [batch, heads, seq_q, seq_k] = query @ torch.transpose(key, -1, -2)
9     biased = scores + rel_pos_bias
```

[Inferred shapes] reveal the bug:

- The function computes attention scores and adds a relative position bias.
- **Mismatch:** `seq_k` ≠ `seq_q`.
- Scores have shape [batch, heads, seq_q, seq_k], while `rel_pos_bias` has shape [seq_k, seq_q].
- The bias must be transposed before addition.

How does it work?

- Lower from Python AST to a control flow graph over basic blocks containing `target = expr` statements
- Check constraints and infer shape of each `expr` via hand-written + signature-informed models
- Make assumption that symbolic dimensional variables are unique and perform equality checks via polynomial canonicalization + syntactic equality check

Authors' Contact Information: Noah Rousell, Brown University, Providence, Rhode Island, noah_rousell@brown.edu; Praccho Muna-McQuay, Brown University, Providence, Rhode Island, praccho@brown.edu.

1 Introduction

If you ask any machine learning practitioner about the most notorious error they encounter when using their favorite Python tensor library, the most likely answer is the dreaded shape mismatch. As PyTorch users ourselves, we've wondered why there isn't a go-to tool to infer the shape of `Tensor` s throughout the functions we write. While several works [2] [4] have tackled this problem, the fact that the vast majority of people don't use such tools already signals that the existing formalisms have not translated into tools that are intuitive, reliable, and convenient enough for real-world development. We want to change that. In this paper, we present a proof-of-concept static analysis to infer shapes and catch shape mismatch errors in PyTorch code.

The greatest obstacle in doing this type of analysis in Python is its dynamism, and a lack of any static typing, let alone our `Tensor` shapes. For example, considering the following snippet.

```
1 def linear(x, W, b):
2     x = torch.matmul(x, W)
3     x = x + b
4     return x
```

While we can infer, by virtue of being passed to `torch.matmul`, that `x` and `W` ought to be of type `Tensor`, we know nothing about their shapes or the shape of the result! How can we statically know that the matrix multiply, or even the addition of `x + b` that follows, are valid operations that won't lead to a runtime error? The answer is shape annotations, specifically around our functions parameters, and potential return shape.

```
1 def linear(x: [B, in],
2            W: [in, out],
3            b: [out]) -> [B, out]:
4     x = torch.matmul(x, W)
5     x = x + b
6     return x
```

If the signature of `torch.matmul` over two-dimensional matrices could appropriately express the constraint that the first operand's last dimension and second operand's first dimension should be the same, then we could verify $\text{in}_x = \text{in}_W$ are indeed the same dimension, and this operation could be deemed valid. The signature for `torch.matmul` would also express the shape of the output in terms of variables from the shape of the input, and we could infer that after the assignment on line 4 that `x` would now have shape [B, out]. Similarly, with a way of resolving the broadcast that happens with the addition with `b`, we would find that [B, out] and [out] indeed broadcast, and produces a resultant `Tensor` with shape [B, out].

What complicates things however is tracking the flows of dimensional variables, especially when new `Tensors` come into inception using them. The following is a common pattern in PyTorch code, where we extract the shape from one `Tensor` to produce a new one.

```
1 def uninteresting_mask(x: [H, W]) -> [H, W]:
2     h, w = x.shape
3     mask = torch.ones((h, w))
4     x = x * mask
5     return x
```

We cannot then just reason in terms of `Tensors` with shapes consisting of dimensional variables, but also take care to track the flows of these dimensional variables out of tensors. In the above when we create a `Tensor` using `torch.ones` with passed locals `(h,w)` as the shape argument, we better be able to say that its resultant `Tensor` will have shape [H, W]. Otherwise,

we would be unable to conclude that the element-wise multiplication that follows on line 4 is shape-safe.

One more complicating factor is reshapes of `Tensor`s, as in the following.

```
1 def flatten_image(x: [B, H, W]) -> [B, HW]:
2     b = x.shape[0]
3     flattened = x.reshape((b, -1))
4     return flattened
```

To resolve the resulting shape, one would want to allow certain dimensions to be expressed as *dimensional expressions* over dimensional variables, such as the product H·W that appears in the resulting shape. For example, reshaping a tensor of shape [B, H, W] using arguments `(b, -1)` requires inferring the unspecified second dimension (just as PyTorch does at runtime). Its size must equal the contraction of the remaining dimensions, i.e. H·W.

The preceding examples emphasize the mechanisms which exist in common PyTorch patterns, and that a good shape inference would have to perform.

1.1 Related Work

The most similar project currently in active use is jaxtyping, which enables users to annotate the shapes of tensor parameters, enabling runtime typecheckers to check shapes on function entry [3]. Despite this being a runtime tool, its popularity demonstrates the importance of solving this problem.

PyTea is a static analyzer for python PyTorch programs, aimed at catching shape errors [2]. It differs from our analysis in that it relies on an SMT solver to check constraints. We hypothesize that most PyTorch programs do not require the path specificity of symbolic execution, and without this complexity devise an eager constraint-checking scheme as an alternative to dispatching constraints to an offline SMT solver.

2 Methods

2.1 Intermediate Representation

We built off the python parser from the RustPython project, before lowering to a simple intermediate representation (IR) more amenable to analysis [1].

The IR for a single python function consists of a directed graph over basic blocks, each of which contain a sequence of statements and terminating instruction. Each statement consists of an expression e, as well as an optional target path π to assign to. We model expressions as paths, constants, binary operators, functions, method calls, tuples, and indexing operations. This is formalized in Figure 1.

2.2 Analysis

2.2.1 Semantics. The analysis is a forward flow analysis that tracks the flow of Variables v inside of functions, and uses function signatures to check constraints and perform inference at callsites. Variables v can represent dimensional variables v, tensors $T[d_1, ..., d_n]$, tuples $(v_1, ..., v_n)$, as well as $\bot$, and $\top$. Dimensional variables represent arithmetic expressions; the leaves are either symbolic (as the B and C are in [B, C]) or a concrete number n. This is formalized in Figure 2.

We rely on user annotations to determine the shapes of parameters and return values of functions and apply our flow analysis inside of functions, tracking how shapes propagate through their contents.

At each location l in the body of a function, we have a domain $\sigma^{l}_{\text{in/out}} : \pi \mapsto \{\overline{v}\}$ mapping from locals with path π to the set of possible variables v the path π may be bound to. In a function body with no control flow, this would (most likely) be a singleton set, since each

$$\begin{aligned}
&\text{Symbol } \alpha \qquad \text{Number } n \qquad \text{String } s \\
&\text{Path } \pi ::= s \mid s.\pi \\
&\text{Constant } c ::= \text{None} \mid \text{Bool} \mid \text{String} \mid \text{Int} \mid \text{Float} \\
&\text{Expr } e ::= \pi \mid c \mid e_1 \oplus e_2 \mid f(e^*) \mid (e_1, \ldots, e_k) \mid e_1[e_2] \\
&\text{Instruction } \mathcal{I} ::= (\epsilon \mid \pi) := e
\end{aligned}$$

Fig. 1. Expression-level syntax.

$$\begin{aligned}
&\text{Symbol } \alpha \qquad \text{Number } n \qquad \text{Location } l \\
&\text{DimVar } d ::= \alpha \mid n \mid d_1 + d_2 \mid d_1 \cdot d_2 \\
&\text{Variable } v ::= \top \mid d \mid T[d_1, \ldots, d_n] \mid (v_1, \ldots, v_k) \\
&\text{Domain } \sigma ::= \pi \mapsto \{\overline{v}\}
\end{aligned}$$

Fig. 2. Analysis-domain syntax.

assignment to a local should fully define its shape for a given location and the ones that follow it (unless reassigned). At the very first location in a function body $l = 1$, we can initialize σ_{in}^1 from the shape annotations we require on our function parameters.

Our analysis produces σ_{in}^l as

$$\sigma_{\text{in}}^l = \bigsqcup_{i \in \text{pred}(l)} \sigma_{\text{out}}^i$$

where we define the join operator $\sqcup$ as simply the union across the mapped sets for each local π

$$\sigma^l \sqcup \sigma^{l'} = \{\, \pi \mapsto \sigma^l[\pi] \cup \sigma^{l'}[\pi] \}$$

We then produce σ_{out}^l as

$$\sigma_{\text{out}}^l = f(\mathcal{I}_l, \sigma_{\text{in}}^l)$$

using a transfer function $f : \mathcal{I} \times \sigma \to \sigma$. At its core, it must be able to take the right-hand side expression of $(\epsilon \mid \pi) := e$, determine the set of variables it may be, and assign that to the optional target path π in σ. We define the meta-function $\mathcal{E} : e, \sigma \mapsto \{\overline{v}\}$ for this task. The transfer function can then be defined in terms of $\mathcal{E}$:

$$f(\mathcal{I}, \sigma) = \begin{cases} \sigma[\pi \mapsto \mathcal{E}(e, \sigma)] & \text{if } \mathcal{I} = \pi := e, \\ \sigma & \text{otherwise.} \end{cases}$$

The function $\mathcal{E}$ is the crux of our analysis, and its definition is the subject of the the following sections.

An assumption implicit in this transfer function is that *procedure calls may not mutate existing references* to `Tensor`s. In other words, evaluating an expression cannot change the shape information associated with any variable already in σ. This is the case for many PyTorch patterns, where functions often produce new references to the same underlying memory (i.e. reshapes) yet leave the original `Tensor`'s shape binding unchanged. Under this restriction, all shape effects are localized to the newly introduced path π, simplifying the analysis considerably.

2.2.2 Evaluating Expressions. For some types of expression e, the behavior of $\mathcal{E}$ is fairly straightforward. These simple cases are

$$\mathcal{E}(e,\sigma) = \begin{cases} \sigma[\pi] & \text{if } e = \pi, \\ \{\, \mathrm{n} \,\} & \text{if } e = n \text{ is a constant, } c \in \mathsf{Int}, \\ \{\, \top \,\} & \text{if } e = c \text{ is a constant, } c \notin \mathsf{Int}, \\ \{\, (v_1, \ldots, v_k) \mid v_i \in \mathcal{E}(e_i, \sigma) \,\} & \text{if } e = (e_1, \ldots, e_k), \end{cases}$$

In essence, accessing a path maps to accessing its representation in σ, constant Ints are considered to be potential concrete DimVars, and tuples $(e_1, \ldots, e_n)$ of expressions are turned into the set of tuple-valued variables formed by taking the Cartesian product of their component analyses. These cases require no additional structure: they simply propagate or wrap the information already present in σ without introducing new constraints on dimensional variables.

2.2.3 Models. For the remaining, more interesting, cases, such as binary operators and method calls, we dispatch within a set of `Model`s that define constraints and infer the return shape for each operator in terms of the input dimensional variables.

For example, a `Model` representing a (non-batched) matrix multiplication operator with signature [a, b], [b, c] → [a, c] would check that the second dimensional variable of the first input `Tensor` is equivalent to the first dimensional variable of the second input `Tensor`, and then infer a return shape of rank 2, with the first dimensional variable coming from the first `Tensor`'s first dimensional variable, and the second dimensional variable coming from the second `Tensor`'s second dimensional variable.

In this case, this `Model` can be fully represented by the operator's signature, but there exist more complex cases that require *procedure-specific models*.

Consider the example of a broadcasted addition, there would exist an infinite number of signatures describing the abstract shapes of its inputs (i.e. [B, D], [D] → [B, D] and [B, C, D], [D] → [B, C, D]). Enumerating all such admissible signatures is impossible. Instead, what we need is a *procedure-specific rule* that takes as input the abstract shapes of the arguments at the call site, and constructs the resulting abstract shape. They are moreover responsible for determining any shape mismatch errors that may occur.

In the case of a broadcasted operation between two tensors, this would involve left appending singleton dimensions onto the shorter shape, then ensuring either each dim matches symbolically, or is singleton. If this check fails for any dimension, we found an error! This is precisely the same as the runtime check that PyTorch does, but lifted to the abstract domain so that it operates over symbolic dimensional variables rather than concrete integers.

We have implemented procedure-specific models for many of PyTorch's core operations, such as **`torch`**`.reshape`, **`torch`**`.cat`, broadcasting, and batched matrix multiplication. We handle shape-annotated user functions via the pure signature approach described.

2.2.4 Constraint Checking. Our analysis catches shape errors by introducing equivalence constraints over dimensional variables. Given that our dimensional variables can represent products and sums of of other dimensional variables, equivalence checking is non-trivial.

We make the assumption that symbolic dimensional variables are only equivalent if they have the same symbol. This means that, for example, a-1 ≠ b-1, even though there are valid assignments to a, b that satisfy this constraint. This assumption enables us to check constraints without an SMT solver by converting dimensional variables to a canonical polynomial form and checking for syntactical equivalence.

We believe it is reasonable to expect developers to use the same symbolic dimensional variables in annotations when appropriate. Consider a function that performs matrix multiplication which is initially typed by a developer as ([a, b], [c, d]) → [e, f]. Our analysis will report an error upon observing the `torch.matmul` constraint over the inner dimensions, reporting that b ≠ c (and another error for the return annotation), prompting the user to adjust the signature to ([a,b], [b,c]) → [a,c]. Note that these annotations lead to more readable code, as they fully encode the relations between the dimensional variables of the parameters and return.

3 Evaluation

The current lowering phase and analysis cannot handle Python dictionaries or classes, which are of course plentiful in true PyTorch programs. In lieu of evaluating our analysis on true programs, we have constructed a functional version of the "Hello world" of deep learning, training a Multi-Layer Perceptron (MLP) network on the MNIST dataset. We therefore can only make claims about the efficacy of our analysis on programs written in a functional style, though we note that PyTea handles more complicated objects in its IR (albeit with a significant cost in complexity), giving us optimism about the feasibility of extending our analysis [2].

3.1 Inference on MNIST

The full (correct) MNIST program can be found at this link, and our analysis's inference at this link (in our IR format); we're including snippets here due to space constraints.

```
def linear(x: [batch, in], weight: [out, in], bias: [out]) -> [batch, out]:
    return x @ torch.transpose(weight) + bias

def forward(x: [batch, 1, 28, 28], w1: [hidden, 784], b1: [hidden], w2:
    [hidden, hidden], b2: [hidden], w3: [classes, hidden], b3: [classes]) ->
    [batch, classes]:
    x: [batch, 784] = torch.view(x, (x.shape[0], -1))
    x: [batch, hidden] = torch.nn.functional.relu(linear(x, w1, b1))
    x: [batch, hidden] = torch.nn.functional.relu(linear(x, w2, b2))
    x: [batch, classes] = linear(x, w3, b3)
    return x
```

In the snippet above, the added annotations to assignments come from our analysis. In this simple case, the analysis is able to infer and check the shape of variables at each location.

In the core training loop below, shape inference is waylaid upon entry into the loop, as our analysis is ignorant of the value of `i`. This highlights a large potential issue: even though our analysis could continue to infer the shapes of `images`, `labels`, and `outputs` if it knew the value of `i`, the introduction of a single top element can have a cascading effect. In this specific case, both `range` iterators are simple enough that we could unroll or otherwise statically analyze them to determine an appropriate value for `i`.

```
def init_weight_matrix(in_features: [in], out_features: [out]) -> [out, in]: ...
    # body elided
def init_bias_vector(out_features: [out]) -> [out]: ... # body elided

def main():
    ... # hyperparameter declarations
    epochs: 10 = 10

```

```
    w1: [256, 784] = init_weight_matrix(input_size, hidden_size)
    b1: [256] = init_bias_vector(hidden_size)
    w2: [256, 256] = init_weight_matrix(hidden_size, hidden_size)
    b2: [256] = init_bias_vector(hidden_size)
    w3: [10, 256] = init_weight_matrix(hidden_size, num_classes)
    b3: [10] = init_bias_vector(num_classes)

    train_images: [1000, 1, 28, 28] = torch.zeros(num_train, 1, 28, 28)
    train_labels: [1000] = torch.randint(0, num_classes, (num_train))
    test_images: [200, 1, 28, 28] = torch.zeros(num_test, 1, 28, 28)
    test_labels: [200] = torch.randint(0, num_classes, (num_test))

    optimizer = torch.optim.Adam((w1, b1, w2, b2, w3, b3), lr=learning_rate)
    num_train_batches: 16 = num_train + batch_size - 1 // batch_size
    for epoch in range(1, epochs + 1):
        total_train_loss = 0.0
        for i in range(num_train_batches):
            start = i * batch_size
            end = min(start + batch_size, num_train)
            images = train_images[start:end]
            labels = train_labels[start:end]

            optimizer.zero_grad()
            outputs = forward(images, w1, b1, w2, b2, w3, b3)
            loss = torch.nn.functional.cross_entropy(outputs, labels)
            loss.backward()
            optimizer.step()
```

3.2 Catching subtle bugs

Below is a function that computes attention and adds a relative position bias to the score matrix. This function has a bug on line 14. Our analysis prints out the inferred shapes through line 13, and reports "mismatched dimension, seq_k ≠ seq_q."

Without the inlaid inferred shapes, it's tricky to see that this function has a bug in it, and where it comes from. With the inferred types and error message, we notice that `scores` on line 13 has shape [.., seq_q, seq_k], which doesn't broadcast with the [seq_k, seq_q] shape of `rel_pos_bias`; this makes it easy to see that `rel_pos_bias` needs to be transposed!

```
def relative_attention_cross(x_q: [batch, seq_q, d_model], x_kv:
    [batch, seq_k, d_model], W_q: [d_model, heads, d_k], W_k:
    [d_model, heads, d_k], W_v: [d_model, heads, d_v], rel_pos_bias:
    [seq_k, seq_q]):
    batch: batch = x_q.shape[0]
    seq_q: seq_q = x_q.shape[1]
    seq_k: seq_k = x_kv.shape[1]
    d_model: d_model = x_q.shape[2]
    heads: heads = W_q.shape[1]
    d_k: d_k = W_q.shape[2]
    d_v: d_v = W_v.shape[2]
    query: [batch, heads, seq_q, d_k] = torch.transpose(torch.reshape(x_q @
    torch.reshape(W_q, (d_model, -1)), (batch, seq_q, heads, d_k)), 1, 2)
    key: [batch, heads, seq_k, d_k] = torch.transpose(torch.reshape(x_kv @
    torch.reshape(W_k, (d_model, -1)), (batch, seq_k, heads, d_k)), 1, 2)
```

```
    value: [batch, heads, seq_k, d_v] = torch.transpose(torch.reshape(x_kv @
    torch.reshape(W_v, (d_model, -1)), (batch, seq_k, heads, d_v)), 1, 2)

    scores: [batch, heads, seq_q, seq_k] = query @ torch.transpose(key, -1, -2)
    biased = scores + rel_pos_bias
    weights = torch.softmax(biased, dim=-1)
    out = weights @ value
    return out
```

References

[1] [n. d.]. RustPython parser. https://github.com/RustPython/Parser

[2] Ho Young Jhoo, Sehoon Kim, Woosung Song, Kyuyeon Park, DongKwon Lee, and Kwangkeun Yi. 2021. A Static Analyzer for Detecting Tensor Shape Errors in Deep Neural Network Training Code. (2021). doi:10.48550/ARXIV.2112.09037

[3] Patrick Kidger, Roman Knyazhitskiy, Brent Yi, Nathan Simpson, Oliver Woodman, Alex Ford, Nima Shoghi, Patrick Kunzmann, Peter Hawkins, Peter Roelants, Sergei Lebedev, Yuxuan Jiang, Zac Cranko, Raffaello Baluyot, Daniel Ward, Eugene Brevdo, jianlijianli, vincentlo-a, Nick Groszewski, Nathaniel Starkman, Matthew Johnson, Martí Zamora, Kevin P Murphy, Jérome Eertmans, Joao Pedro Araujo, Dangyi Liu, Artur Chakhvadze, Andy Rock, Alex Fan, and Afroz Mohiuddin. 2025. patrick-kidger/jaxtyping. https://github.com/patrick-kidger/jaxtyping. https://github.com/patrick-kidger/jaxtyping

[4] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs (Artifact). 6:1–6:3 pages. doi:10.4230/DARTS.6.2.6