Dealing with GuardOnDataDependentSymNode errors

Feb 7, 2024
Edward Yang

If you are working on PT2 support for models that have data-dependent control flow, e.g., via item(), tolist(), nonzero(), etc., a likely error you will run into is this one:

```
torch.fx.experimental.symbolic_shapes.GuardOnDataDependentSymNode: Could
not guard on data-dependent expression Eq(u2, -1) (unhinted: Eq(u2, -1)).
(Size-like symbols: none)

Potential framework code culprit (scroll up for full backtrace):
  File "/data/users/ezyang/a/pytorch/torch/_prims_common/__init__.py", line
855, in infer_size
    if d == -1:

For more information, run with TORCH_LOGS="dynamic"
For extended logs when we create symbols, also add
TORCHDYNAMO_EXTENDED_DEBUG_CREATE_SYMBOL="u2"
If you suspect the guard was triggered from C++, add
TORCHDYNAMO_EXTENDED_DEBUG_CPP=1
For more debugging help, see
https://docs.google.com/document/d/1HSuTTVvYH1pTew89Rtpeu84Ht3nQEFTYhAX3Ypa
_xJs/edit?usp=sharing
```

The root cause of errors like this is that somewhere in PyTorch (or potentially user code) we are trying to convert a symbolic quantity (e.g., u2 == -1) into a concrete one (e.g., False) so that we can branch on it or pass it to a subsystem that doesn't support symbolic reasoning. Ordinarily (i.e, when data dependent sizes are not involved), we know the concrete value (the backing hint) and can just give it to you, installing an appropriate guard to make sure we don't use this compilation result when the concrete value would be different. But with data dependent quantities, we do not know what the true value is (it is, after all, data dependent) and now we are stuck.

Fortunately, it is often possible to rewrite your model (e.g., by adding torch._check or torch._check_is_size tests) so that you can bypass these problems. The purpose of this doc is to teach you how to do this.

If you like this doc, you may also like The dynamic shapes manual and PendingUnbackedSymbolNotFound

Data-dependent puzzlers

If you like learning more by doing as opposed to reading, we have a series of exercises at https://www.internalfb.com/intern/anp/view/?id=5330476 which walk you through a number of common situations with data-dependent errors and challenge you to find fixes for them. It's a good way to get your feet wet with a fast feedback loop without having to deal with a full on model. These are publicly available at https://github.com/ezyang/data-dependent-shape-puzzles/

Variations of the error

Could not guard on data-dependent expression

This means we tried to extract a concrete **boolean** from a relational expression like u0 == 0 or u0 > 10. In these cases, it may be possible that the value is *always* True or False, in which case we can fix tracing by informing the symbolic reasoning which way we should go.

This should be distinguished from...

Could not extract specialized integer from data-dependent expression

This means we tried to extract a concrete **integer** from an expression; it's pretty common to see just u0 here. There are two typical causes of this:

- You *actually* need the integer because you're going to do some sort of control flow; e.g., you want to loop u0 times, or you want to index into the u0'th element of a Python list. These cases are unsolvable; you want to graph break (e.g., torch._dynamo.graph_break()) before the dynamic access and try for compilation at a later point in time.

- You are *overspecializing* for some reason. For example, we might have some poorly written C++ code that hasn't been ported to SymInt; this will force a specialization even in principle the C++ code could be symbolically traced. In this case, you want the same playbook to eliminate specialization on plain backed SymInts; see The dynamic shapes manual for some possible moves.

Tools of the trade

There are a few important functions which you are likely to use. In order of importance:

torch._check(cond, msg_fn)

Example usage:

```
torch._check(x.size(0) == y, lambda: f"size mismatch: {x.size(0)} != {y}")
```

Semantically, this is equivalent to:

```
if not cond:
    raise RuntimeError(msg_fn())
```

However, there are two important differences:

- This test will *always* succeed at compile time, even if cond involves unbacked SymInts. Instead of installing a guard, we will install a deferred runtime assert which will test that this condition is actually true at runtime. We don't need to know if cond is True or False (like regular conditionals), because at compile time we can just assume that an error will not be thrown.

- In some situations, the condition teaches our symbolic reasoning facts about your unbacked SymInts, which it can use to discharge later conditionals. Our system currently supports learning the following facts from conditions:

    - If you perform an equality test u0 = RHS, we will attempt to eliminate u0, replacing all occurrences of it with RHS. We will ALWAYS do this if RHS doesn't contain any unbacked symbols (since eliminating unbacked symbols is good--you can't have a GuardOnDataDependentSymNode if the unbacked symbol goes away)~~, and we will try to eliminate unbacked SymInts if the situation is not too complicated (in particular, u0 = u1 will eliminate one of these unbacked SymInts.)~~

    - In the equality test above, even if we are not able to eliminate u0, we can refine its value range. The value range specifies what the set of possible values for a variable are. By default, size-like unbacked SymInts have a value range of [0, Inf]; if you assert it is equal to an expression with a refined value range, say [2, 20], then u0's value range will be updated to [2, 20]. We also have limited support for propagating value ranges in reverse.

    - If you perform a boolean test f(u0), we will remember that this expression always evaluates to True, and if you evaluate an expression that contains this expression, we will substitute it with True. We also support some limited reasoning on logically equivalent statements; e.g., if you torch._check(u0 < 4), we will also know that u0 >= 4 evaluates to False, and so performing a test like this in a normal non-check conditional will go through fine.

    The exact symbolic reasoning our system supports is not well defined, and we reserve the right to make it stronger in the future. However, we do have tests for various uses of reasoning, and we ensure that changes to our reasoning will not break these uses.

Our decompositions / meta implementations of functions in the PyTorch API make ubiquitous use of torch._check, so chances are that when you use an unbacked SymInt with these APIs, we are already learning things about your SymInts without you needing to explicitly spell things

out. You can find out exactly what we are learning by running your program under TORCH_LOGS=dynamic and looking for “runtime_assert” log messages. However, sometimes we will hit a non-torch._check conditional on a fact that you, as a user know, but we, the framework, do not. This is the typical use case for inserting a new torch._check.

In C++, the corresponding version of this is TORCH_SYM_CHECK. However, you must take care to actually do a symbolic relational test (the regular == on SymInt will immediately guard on the boolean). For example, torch._check(x.numel() == y.numel()) would translate into TORCH_SYM_CHECK(x.sym_numel().sym_eq(y.numel())), using the sym_eq method on SymInt to perform a symbolic test that returns a SymBool.

torch._check_is_size(size) / guard_size_oblivious(cond)

Example usage:

```
u0 = y.item()
torch._check_is_size(u0)
```

Semantically, this is equivalent to:

```
if u0 < 0:
    raise RuntimeError("u0 is not a size")
```

However, like torch._check, there are some important differences:

- Like torch._check, this test will always succeed at compile time, and we will learn the fact that u0 >= 0. In particular, this will refine the value range on u0 so that it is [0, Inf] rather than [-Inf, Inf].

- This also marks u0 as *size-like*. Size-like unbacked SymInts behave identically to their regular counterparts, except for one crucial difference: when they are involved in a boolean expression that is evaluated using guard_size_oblivious, for the purposes of evaluating that expression only, we will assume that they cannot equal zero or one (in other words, their value range is temporarily set to [2, Inf].) For example, if we perform a (non-torch._check) conditional on u0 == 1 (to see if broadcasting will occur), when u0 is size-like, we will evaluate this to False rather than throw an error.

For example: guard_size_oblivious(u0 == 1) will always return False when u0 is size-like.

Marking unbacked symbols as size-like is crucial for using them in contexts where tensor sizes are expected, because PyTorch internals often do many conditions on whether or not sizes are zero or one, to handle various special cases related to empty / single element tensors. In fact, if you pass an unbacked symbol to a conventional factory function like torch.empty, we will

automatically mark it as size-like for you. However, sometimes we are not able to infer that a quantity is size-like. For example, arguments to Tensor.view cannot be inferred to be size-like, as -1 is also a valid argument. You would need to explicitly torch._check_is_size an unbacked SymInt before passing it to view.

Similarly, if you are in PyTorch framework code and you need to perform a test on a size to see if it is 0 or 1, you will typically want to wrap this test in guard_size_oblivious, to instruct that size-like unbacked SymInts can be assumed to not pass this test. It is a bit difficult to say exactly when it is OK to do this, but in general, most framework code has logic for >= 2 case which would work perfectly fine for 0/1 case, so if patching something in PyTorch framework to guard_size_oblivious fixes your problem, it's probably OK. A case when this would be *clearly wrong* is if you actually need to do something different for the 0/1 case even at runtime; e.g., a hand-tracking application testing the number of hands it has detected. So in general, it is usually *not* OK to use guard_size_oblivious in user code!

This can be done in C++ with TORCH_GUARD_SIZE_OBLIVIOUS(u0.sym_eq(0)), for example.

torch._check_is_size(size, max=upper_bound)

NOTE: This is NEW as of https://github.com/pytorch/pytorch/pull/144471 Jan 14, 2025

This is semantically equivalent to torch._check(size <= upper_bound), but it also has different semantics under guard_size_oblivious: we will assume that size < upper_bound. This only works when upper bound is an integer constant (so you get normal semantics if upper_bound is a symbolic expression; we can potentially fix this without too much work.)

https://github.com/pytorch/pytorch/issues/120288

torch._constrain_as_value / torch._constrain_as_size

These are much more niche APIs that are effectively equivalent to torch._check/torch._check_is_size, except they also let you adjust the value range of the variable by providing a min/max value. In recommendation models, you are unlikely to be able to solve GuardOnDataDependentSymNode errors in this way.

Although constrain_as_value looks like an attractive way to specify that a variable should be in-bounds of another tensor, in practice it is often unusable, because value ranges only support constant bounds, and it is pretty common that the tensor you want to index into has a (backed) symbolic dimension s0. Passing its size as the max value for a value range will force it to be specialized, which is often not what you want. Instead, if necessary, you should manually discharge range checks by torch._check()'ing appropriate expressions based on what errors you see.

What a fix looks like

There are a number of common ways you are going to resolve a problem like this. We describe the most common ones here.

It's unfixable

Sometimes, it's actually unfixable.

```
i = x.item()
if i > 4:
  return x * 2
else:
  return x + 3
```

If the user code is legitimately branching on a data-dependent value, it is literally impossible to trace as is and you'll have to do something else, maybe use a torch.cond.

Another common pattern is if you do this:

```
return self.mlps[x.item()]
```

Where self.mlps is a Python list / ModuleList. Once again, you are branching on a data-dependent value. Here, the easiest fix is to induce a graph break before indexing.

u0 is a size, but we don't know it

Some guards fail on tests that essentially ask "is this a size", but we don't know it is a size. These fall into two categories:

- Regular tests like "u0 >= 0" or "u0 != -1" that are unconditionally true for sizes. Adding a torch._check_is_size(...) on the relevant size will instruct that these tests are true. (However, this is typically uncommon, because if the test in question is for error checking, we can infer that the quantity must be true, because we would error otherwise. One important exception are APIs which accept both sizes as well as -1; it is necessary for the user to tell us that the input data dependent quantity cannot possibly be -1, as if it could be, something unusual would happen.) Example: https://github.com/pytorch/pytorch/pull/107788
    - Sometimes, you can refactor an error checking API to split a logical disjunction of conditionals into separate conditionals. If you can do so to get a single torch._check(x == y) statement, this will make it possible to automatically generate a deferred runtime assert. Example:

https://github.com/pytorch/pytorch/pull/110979

- Tests like u0 == 0 or 1 which are not always true for sizes, but for which our choice doesn't really matter (it is handling an edge case such as what to do with an empty tensor, or it is testing for broadcasting but we want to assume broadcasting is not occurring.) We can resolve these situations with two ingredients: first, the guard itself must be evaluated via guard_size_oblivious, which says "for all size-like ints, assume they cannot equal zero or one, and I promise that if they do equal zero/one something reasonable will happen." Second, the symbols themselves must be marked size-like (either inferred because they were passed to tensor factory functions, or explicitly specified with torch._check_is_size(...)). Many examples of making guards size oblivious in https://github.com/pytorch/pytorch/pull/118579

Sometimes, these tests can occur in C++. We have corresponding C++ APIs for these tests, but it can be a little more difficult to localize the problem as you do not get a useful backtrace by default.

u0 is actually equal to u1, but we don't know it

Multiple unbacked SymInts can actually known to be equal at compile time:

```
i0 = x.sum().item()
i1 = x.sum().item()
return torch.randn(i0) + torch.randn(i1)
```

If there is a torch._check(i0 == i1) somewhere (in the example above, this test will happen inside the shape checking rule for addition), we will automatically unify the two unbacked SymInts and report them as equal. However, sometimes there is no such assert, and you have to explicitly add an assert to get it to work. Example: https://github.com/pytorch/pytorch/issues/111950

We recently fixed some bugs in PyTorch where we incorrectly reallocate unbacked SymInts in framework code, and then failed to discover that they are equal. We have fixed many of them but this may also potentially be the cause of a problem. Example: https://github.com/pytorch/pytorch/pull/117862 Note that if we allocate an unbacked SymInt, but then immediately set it equal to another one, these are benign (and not so easy for us to eliminate entirely from the framework).

u0 is a Tensor

Another reason you might be overallocating unbacked SymInts is that you are passing around a Tensor and relying on implicit conversion to int. For example, most functions which accept an int, also accept a Tensor, and will automatically call item() on the int argument. It is worthwhile to study TORCH_LOGS=dynamic and decide if you expect to see all the unbacked SymInts you see, or if there are too many. When this happens, a new SymInt will be allocated on the line where a PyTorch function is called

(This is less likely to cause problems now, because we memoize the return value of t.item(), so you'll consistently get the same unbacked symint if you call it multiple times)

You are overspecializing

Consider the following code in non-strict export mode:

```
u0 = x.sum().item()
return y[:u0]
```

This will fail trying to evaluate u0 (sic; not a boolean expression). The reason for this is that when a SymInt is used inside a Python slice directly (e.g., not with Dynamo), Python will actually force the integer to be specialized, and fail if it is unbacked.

This can be fixed by rewriting the program not to specialize. For the particular example above, you can fix it by rewriting the code not to use slices:

```
u0 = x.sum().item()
return y.narrow(0, 0, u0)
```

Example: https://github.com/pytorch/pytorch/issues/111950

Use lengths instead of offsets

When working with variable sequence lengths, it is common to have tensors representing the lengths or offsets of the sequences involved. For example, if I have values = [[1, 2, 3], [4, 5], [6, 7, 8, 9]], I might have lengths = [3, 2, 4] and offsets = [0, 3, 5, 9]. The two representations are interconvertible but if you are working with them as integers (by calling lengths.tolist()), it is better to call tolist on lengths, not offsets.

The reason for this is if you perform a torch.split() on your values tensor, we need to create Tensors for each sub-sequence involved, e.g., a size 3, 2 and 4 tensor. If you have unbacked SymInts for sizes, these sizes become u0, u1 and u2; you indicate that they are size-like and you are done. But if you have unbacked SymInts for offsets, they become u1 - u0, u2 - u1, u3 - u2. This is a pain, because you cannot conveniently mark these quantities as size-like and there

are problems. Because it is relatively simple to write code to do either lengths or offsets, you should prefer lengths.

More discussion at https://github.com/pytorch/pytorch/issues/119468

How to diagnose your problem

Step 1: Look at the potential framework culprit (aka the Python backtrace)

The exception has a backtrace, and often it will tell you what the problem is. Since PT2 backtraces are quite long, the error message will also give you its best guess for what the framework culprit is, e.g.,

```
Potential framework code culprit (scroll up for full backtrace):
  File "/data/users/ezyang/a/pytorch/torch/_prims_common/__init__.py", line
855, in infer_size
    if d == -1:
```

Look at the condition in question.

- Does it make sense that this is triggering a guard on data-dependent symbol?
- Should we know if the quantity in question is size-like (the exception tells you which symbols are size-like; if a symbol is not in the list, we are assuming it could be an arbitrary integer)?
- If the equation is between two distinct symbols, should we know that these are actually equal?
- If all the symbols are size-like, but the equation involves 0 or 1, are we missing a guard_size_oblivious wrapper on the equation? (Don't forget that to do a guard_size_oblivious between two size tuples, you have to use sym_eq and not regular equality.)

In this particular example, we are testing if d (ostensibly the data dependent value) is -1; if d were a size, we would know it was definitely non-negative. So this would suggest that we are missing a torch._check_is_size somewhere. Or if d is already size-like, but numel() == 0 is failing, it sounds like you need to wrap it in guard_size_oblivious.

TORCH_LOGS=dynamic and the user stack trace are in practice important for understanding how exactly to fix the problem, because they will help you determine how exactly you should change the user program.

```
[INFO] create_unbacked_symint u0 [-9223372036854775808,
9223372036854775807] (w.py:40 in custom_op_meta)
```

This log message tells you where (w.py:40) this particular unbacked SymInt was allocated. A given unbacked SymInt may get allocated multiple times, so you may need to keep track of equalities on them:

```
[INFO] set_replacement u1 = u0 (trivial_lhs) ValueRanges(lower=0,
upper=9223372036854775807, is_bool=False)
```

Step 2: Look at the C++ backtrace

If the potential framework code culprit is completely uninformative, your guard may potentially be happening in C++. You can force a best effort C++ backtrace by running with TORCHDYNAMO_EXTENDED_DEBUG_CPP=1. This will give you an extremely *long* C++ backtrace with Python frames, CPython frames and C10/ATen/libtorch proper frames all interspersed together. You're looking for some symbols in at:: or c10:: namespace that look like kernel specific code (likely related to the kernel that was being executed as per the Python backtrace.) If you're running on a non debug build of PyTorch, due to inlining, you may be missing frames, so you will also have to do some sleuthing in the source code to find the source code location. Example: https://github.com/pytorch/pytorch/pull/118579

Here is an example C++ backtrace from a debugging session at https://fb.workplace.com/groups/6829516587176185/posts/6829896033804907

```
[2024-02-08 08:20:45,259] torch.fx.experimental.symbolic_shapes: [INFO]   File
"./buck-out/v2/gen/fbcode/aa1e544c619c762d/caffe2/__gen_aten__/out/RegisterComposit
eImplicitAutograd.cpp", line 2025, in at::(anonymous namespace)::(anonymous
namespace)::wrapper_CompositeImplicitAutograd_Tensor_narrow(at::Tensor const&,
long, at::Tensor const&, c10::SymInt)
214[2024-02-08 08:20:45,259] torch.fx.experimental.symbolic_shapes: [INFO]   File
"./fbcode/caffe2/aten/src/ATen/native/TensorShape.cpp", line 1410, in
at::native::narrow_tensor_symint(at::Tensor const&, long, at::Tensor const&,
c10::SymInt)
215[2024-02-08 08:20:45,259] torch.fx.experimental.symbolic_shapes: [INFO]   File
"./buck-out/v2/gen/fbcode/aa1e544c619c762d/caffe2/__gen_aten__/out/core/TensorMetho
ds.cpp", line 52, in long at::Tensor::item<long>() const
216[2024-02-08 08:20:45,259] torch.fx.experimental.symbolic_shapes: [INFO]   File
"./fbcode/ATen/core/TensorBody.h", line 4274, in at::Tensor::item() const
```

Here, you can see that at::native::narrow_tensor_symint calls into item (which in our case is what triggers the guard on data-dependent SymNode). You can then modify the C++ code to avoid

specializing, or perhaps you weren't supposed to be in this C++ code in the first place (in this case, start was not expected to be a Tensor, and modifying this fixed the problem.)

By the way, the C++ backtraces are *much* better in fbcode than in OSS (although I find I usually get enough information from the OSS trace), so if you are having trouble understanding your C++ backtrace, consider trying with an fbcode build.

Common issues

torch.tensor_split: Could not guard on data-dependent expression u1 < 0

Alas, torch.tensor_split(tensor, indices_tensor) will basically always fail to trace. This is because tensor_split *technically* supports negative values in the indices tensor, and we have no idea how to generate a generic kernel that works both with positive and negative indices. If you know that int_indices must not be negative, you can rewrite your code to:

```
int_indices = indices.tolist()
for i in int_indices:
  torch._check_is_size(i)
return torch.tensor_split(tensor, int_indices)
```

TorchScript compatibility

If the code to be modified must work under TorchScript, you can gate calls to torch._check_is_size and co with:

```
if not torch.jit.is_scripting() and is_torchdynamo_compiling():
  torch._check_is_size(blah)
```