

What happens when torch.compile goes wrong?

torch.compile offers a clear value proposition: add a single line to your code, and speed up your model with no extra work. But it's a complex system that can fail in unintuitive ways. The immediate symptom (a graph break or unexpected recompile) may be far from the cause, especially for larger models. Sometimes there's an easy fix, but other times the fix may involve experimental features, rewrites, or other compiler-specific knowledge.

517

graph break
types

885

compiler config
options
dynamo ·
inductor · aot · fx

69

TORCH_LOGS
channels &
aliases

A The easy case: the message has the fix

```
your code
@torch.compile(fullgraph=True)
def f(x):
    y = x.relu()
    print("mean activation:", y.mean())
    return y * 2
```

```
Unsupported: Failed to trace builtin operator
Dynamo does not know how to trace builtin operator
'print' with argument types ['str', 'Tensor']
Hint: If you are attempting to call a logging function
(e.g. 'print'), you can try adding it to
'torch._dynamo.config.reorderable_logging_functions'.
```

✓ fix: do what the hint says

```
torch._dynamo.config.reorderable_logging_functions \
    .add(print)
```

This fix works, but be careful not to mutate the logged tensor!

Surprising recompiles

B Saving results to a dict

```
your code
results = {}

@torch.compile(fullgraph=True)
def f(x, name):
    y = x.cos()
    results[name] = y          # keep for later
    return y

for i in range(10):
    f(torch.randn(3, 3), f"tensor_{i}")
```

```
torch._dynamo hit config.recompile_limit (8)
function: 'f'
last reason: 0/7: name == 'tensor_7' # results[name] = y
FailOnRecompileLimitHit: Hard failure due to fullgraph=True
```

Dynamo treats string inputs as constants, and needs to guard on them. This can lead to surprising recompiles. Without fullgraph=True it's quieter: 8 compiles, then f silently runs eagerly.

✓ fix: skip the bookkeeping while compiling

```
def f(x, name):
    y = x.cos()
    if not torch.compiler.is_compiling():
        results[name] = y
    return y
```

One possible fix is to drop the side effect when compiling. Other possible fixes include moving the write outside of the compiled region or forcing a graph break. These have tradeoffs which may not be obvious to inexperienced users.

C An int attribute that recompiles

```
your code
class Attention(nn.Module):
    def forward(self, x):
        head_dim = x.shape[-1] // self.n_heads
        ... # view / reshape using head_dim
run = torch.compile(lambda block, x: block(x))
for block in [Block(512, 4), Block(512, 8), Block(512, 16)]:
    run(block, x)                # 3 compiles
```

```
$ TORCH_LOGS=recompiles python model.py
Recompiling function run
- 0/0: block._modules['attn'].n_heads == 4
```

A plain int argument would recompile once and then become dynamic automatically. Int attributes on an nn.Module are treated as constants by default, so every head count gets its own graph.

✓ fix: treat module ints like int arguments

```
torch._dynamo.config \
    .allow_unspec_int_on_nn_module = True
```

3 → 2 compiles, or 1 with dynamic=True. Safe here because n_heads only feeds arithmetic; if it picked a branch, the recompiles would be genuine.

Arcane APIs

D Calling a library Dynamo can't trace

Dynamo can only trace Python, not native code. Some Python builtins and common NumPy functions are supported, but in general you need to do some work for Dynamo to understand native code.

```
your code
def forward(self, x):
    y = scipy.special.ellipk(x.cpu().numpy())
    return torch.from_numpy(y).to(x.device, x.dtype)
```

```
Unsupported: Unsupported method call
Dynamo does not know how to trace method '__call__'
of class 'ufunc'
```

✓ fix: wrap it in a custom op (one opaque graph node)

```
@torch.library.custom_op("mylib:ellipk", mutates_args=())
def ellipk(x: Tensor) -> Tensor:
    y = scipy.special.ellipk(x.cpu().numpy())
    return torch.from_numpy(y).to(x.device, x.dtype)

@ellipk.register_fake
def _(x):
    return torch.empty_like(x)

torch.library.opcheck(ellipk, (x,)) # validate contract
```

Custom ops are usually the right choice for numerical code/custom kernels. If you want Dynamo to trace a C function, you can provide a Python implementation with substitute_in_graph (NumPy ufuncs like this one aren't supported).

E Higher-order ops

The rewrite is mechanical but obscure (bodies become pure functions, state becomes an operand tuple): easy for an agent that knows the recipe.

```
loop until convergence → torch.while_loop
# before: data-dependent loop condition breaks the graph
while (x - prev).abs().max() > 1e-4:
    prev, x = x, torch.cos(x)
# after
x, _ = torch.while_loop(
    lambda x, prev: (x - prev).abs().max() > 1e-4,
    lambda x, prev: (torch.cos(x), x.clone()),
    (x, x + 1))
```

debug print inside a module → print HOP

```
# before: graph break on print
print(f"block in: max={h.max():.3f}")
# after: f-string → format string + args
from torch._higher_order_ops import print as hop_print
hop_print("block in: max={:.3f}", h.max())
```

Using AI Agents

AI agents are good at following documentation and applying specific fixes. They can even one-shot more complicated/poorly defined problems, but occasionally go off the rails. To make sure the agents are pointed in the right direction, we develop skills describing the debugging process, general rules about when to apply various fixes, and tradeoffs to explain to the user. We aim to improve the reliability of AI-based debugging, while also reducing token usage and enabling the use of less powerful models.

F The skills

torch-compile-logs	The first step for every problem. Captures TORCH_TRACE + tlparse (or TORCH_LOGS) and explains how to read compile ids, guards, and graphs.
debug-graph-breaks A · D · E prior work by Arsh Zahed	Looks up each break's type and hint category, then removes it: logging configs, the print HOP, leaf_function, nonstrict_trace, disable.
debug-recompilations B · C	Finds the failing guard and decides whether the recompile is genuine or fixable. Fixes with is_compiling(), allow_unspec_int_on_nn_module or dynamic_sources, or recompile limits.
debug-dynamic-shapes	Explains why a dim specialized or why an unbacked size can't be guarded. Fixes with mark_dynamic, mark_unbacked, branchless rewrites, or torch.check.
higher-order-ops E	Rewrites data-dependent branches, loops, and prints as cond, while_loop, scan, or the print HOP so they stay compiled.
custom-ops D	Wraps code compile can't handle (third-party libraries, custom kernels, numerics-sensitive code) as a torch.library op with a fake kernel, checked by opcheck.

G Help us improve!

Hit a graph break, excessive recompiles, or a cryptic torch.compile error? Install the skills and point your agent at it.

```
$ npx skills add meta-pytorch/skills
> My model in train.py recompiles every
step. Figure out why and fix it.
```

Found a case the skills get wrong? Repros, issues, and new skills are welcome.

github.com/meta-pytorch/skills

