Dynamo and How to Introspect It

Mario Lezcano Casado

27th April 2024

The `torch.compile` Stack

`torch.compile` has three main compontents:

- Dynamo: A Python tracer
- AoT Autograd: A second tracer (?!)
- Inductor: The compiler

Introspecting `torch.compile`

`TORCH_LOGS=help`

- `torch.compile` provides the environment variable `TORCH_LOGS`
- Running a PyTorch program with `TORCH_LOGS=help` will show you the most useful options you can pass to it.

Dynamo is a Python tracer

TORCH_LOGS=graph_code

Traces through a Python function given some inputs and records the PyTorch operations that are executed

- The list of executed operations is called a **trace** and it is stored in an **FX graph**.

```python
# ex.py
import torch

@torch.compile
def mse(x, y):
    z = (x - y) ** 2
    return z.sum()

x = torch.randn(200)
y = torch.randn(200)
mse(x, y)
```

```python
# TORCH_LOGS=graph_code python ex.py

def forward(l_x_: torch.Tensor,
            l_y_: torch.Tensor):
    # File: ex.py:6, code: z = (x - y) ** 2
    sub = l_x_ - l_y_
    z = sub ** 2
    # File: ex.py:7, code: return z.sum()
    sum_1 = z.sum()
    return (sum_1,)
```

Dynamo is a linear tracer

TORCH_LOGS=graph_code

- It removes all control flow operators (`if/else`, loops, exceptions...)
- It specialises (*i.e.*, "bakes in") all non-tensor objects (numbers, strings, classes...)

```python
@torch.compile
def fn(x, n):
    y = x ** 2
    if n >= 0:
        return (n + 1) * y
    else:
        return y / n

x = torch.randn(200)
fn(x, 2)
```

```python
def forward(l_x_: torch.Tensor):
    # code: y = x ** 2
    y = l_x_ ** 2

    # code: return (n + 1) * y
    mul = 3 * y
    return (mul,)
```

Dynamo can trace integers symbolically

TORCH_LOGS=graph_code

Static by default: Dynamo bakes in every integer into the graph by default

- If on a subsequent call the value of an int changes, it traces it symbolically
- . . . unless the value is a 0 or a 1. 0 and 1 are always specialised

```python
@torch.compile
def fn(x, n):
    y = x ** 2
    if n >= 0:
        return (n + 1) * y
    else:
        return y / n

x = torch.randn(200)
fn(x, 2)
fn(x, 3)
```

```python
# [...] case n=2 omitted

def forward(l_x_ : torch.Tensor,
            l_n_ : torch.SymInt):
    # code: y = x ** 2
    y = l_x_ ** 2

    # code: return (n + 1) * y
    add = l_n_ + 1
    mul = add * y
    return (mul,)
```

Dynamo's transformations are sound

TORCH_LOGS=graph_code

If it cannot use the previous trace, it will retrace.

```python
@torch.compile
def fn(x, n):
    y = x ** 2
    if n >= 0:
        return (n + 1) * y
    else:
        return y / n

x = torch.randn(200)
fn(x, 2)
fn(x, 3)
fn(x, 6)  # can use n >= 0
fn(x, -3) # retrace!
```

```python
# [...] case n==2 omitted
# [...] case n>=0 omitted

def forward(l_x_ : torch.Tensor,
            l_n_ : torch.SymInt):
    # code: y = x ** 2
    y = l_x_ ** 2

    # code: return y / n
    truediv = y / l_n_
    return (truediv,)
```

Soundness: Guards

TORCH_LOGS=guards,recompiles

Guards are boolean expressions that depend on the inputs of the function

- Guards are created and accumulated at tracing time
- If the guards generated by some inputs are true for a second set of inputs, that means their trace agrees and we do not need to retrace

```python
@torch.compile
def fn(x, n):
    y = x ** 2
    if n >= 0:
        return (n + 1) * y
    else:
        return y / n

x = torch.randn(200)
fn(x, 2)
fn(x, 3)
fn(x, -3)
```

```
GUARDS:  # Case n=2
L['n'] == 2

Recompiling function fn in ex.py:3  # Case n=3
triggered by the following guard failure(s):
L['n'] == 2
GUARDS:
L['n'] >= 0

Recompiling function fn in ex.py:3  # Case n=-3
triggered by the following guard failure(s):
L['n'] == 2    L['n'] >= 0
GUARDS:
L['n'] < 0
```

Completeness: Graph breaks

`TORCH_LOGS=graph_breaks,explain`

Supporting natively all Python is not difficult, it is impossible!

- When dynamo does not understand a construction it graph breaks, that is, it:
  - Stops tracing the current graph
  - Asks CPython to execute that construction
  - Continues tracing the rest of the function into a new graph

```python
@torch.compile
def fn(x):
    y = x ** 2
    print(y)
    return y / 2

x = torch.randn(200)
fn(x)
```

```
Graph break:
call_function BuiltinVariable(print)
              [TensorVariable()]
              {}

from user code at: File "ex.py", line 6, in fn
  print(y)
```

Implementing Dynamo: PEP 523

- Since Python 3.6, Python exposes an API to install **frame evaluators**
- A **frame** is a function and its context (local and global variables)
- PEP 523 was designed to enable the implementation of JITs
- The decorator `@torch.compile` installs the frame evaluator
- `@torch.compile` has full control over every subsequent call to the function

Dynamo as a Bytecode to Bytecode Transpiler `TORCH_LOGS=bytecode`

- When a `torch.compile`'d function is called, Dynamo receives the Python bytecode and the local and global variables
- Dynamo implements CPython interpreter (a stack machine) that simulates the execution of the program. It then records in the FX graph every PyTorch call it finds

Example: Compiling `torch.sin(x+y)`, dynamo transforms the bytecode as:

```
 0 LOAD_GLOBAL      0 (torch)
 2 LOAD_METHOD      1 (sin)
 4 LOAD_FAST        0 (x)
 6 LOAD_FAST        1 (y)
 8 BINARY_ADD
10 CALL_METHOD      1
12 RETURN_VALUE
```

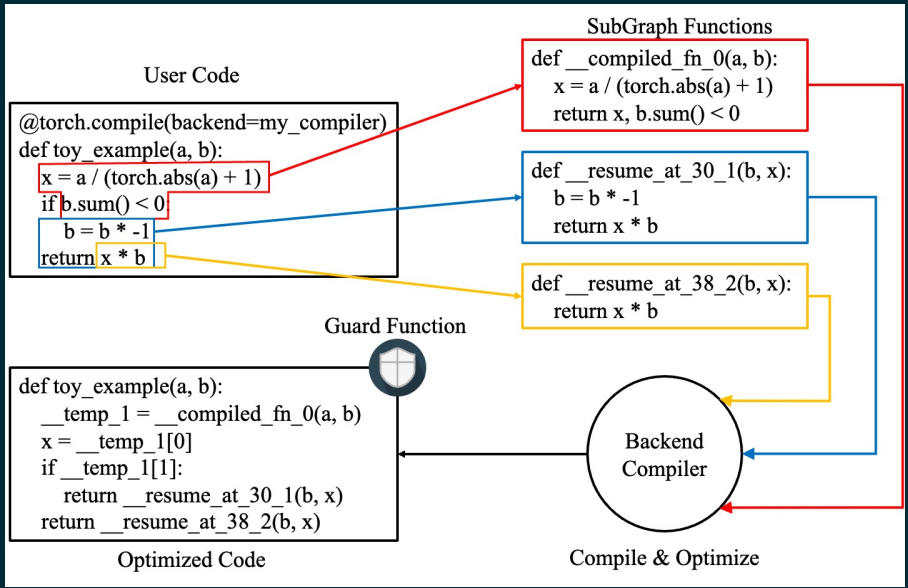
```
 0 LOAD_GLOBAL      2 (__compiled_fn_1)
 2 LOAD_FAST        0 (x)
 4 LOAD_FAST        1 (y)
 6 CALL_FUNCTION    2
 8 UNPACK_SEQUENCE  1
10 RETURN_VALUE
```

Summary of Dynamo's Execution Flow

Assume we call a function with no graph breaks twice in a row with the same arguments.

1. In the first call:
    - 1.1 Dynamo traces the function into an FX graph
        - 1.1.1 The FX graph is compiled by the compiler (Inductor) into efficient CPU / GPU code.
    - 1.2 It rewrites the bytecode of the function so that it simply calls the compiled function
    - 1.3 It gives CPython this new bytecode and asks it to run it
2. In the second call:
    - 2.1 It checks the guards from the first call against the new arguments. Since they are the same arguments as before, they pass
    - 2.2 It asks CPython to run the bytecode associated to those guards

Bytecode Transformation with Graph Breaks



Summary

- Dynamo is a linear Python tracer
- Given some inputs it creates an FX graph with the PyTorch operations it executed
- It specialises all non-tensor values...
- ...but it traces integers symbolically if these change between executions
- It uses guards to decide whether it needs to retrace
- It may fallback to CPython when it finds constructions it does not understand
- It delivers all this implementing a CPython interpreter and performing bytecode to bytecode transformations

Questions?

Extended tutorial at:

https://pytorch.org/docs/main/torch.compiler_dynamo_deepdive.html