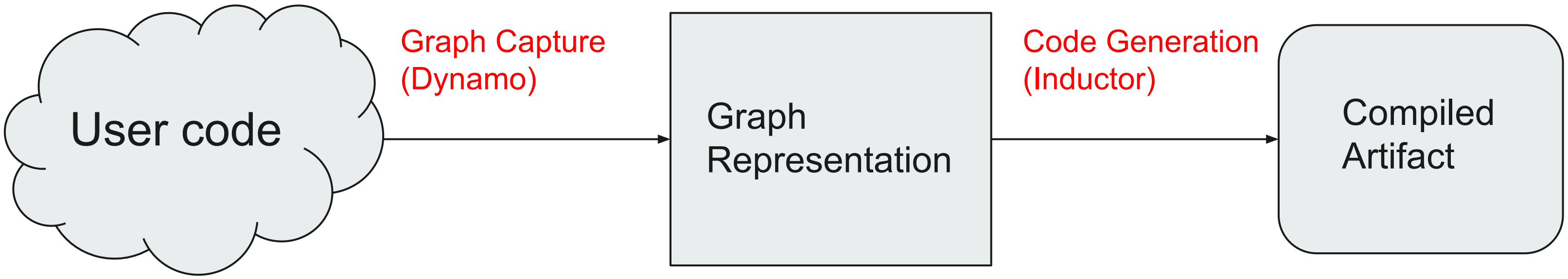
AOTAutograd Overview

Background

Overall compilation flow

1. Capture user code into a graph representation
2. Compile the graph representation into efficient code



Plan

- Problem Statement (what is AOTAutograd solving)
- Automatic Differentation (background)
- PyTorch's C++ autograd engine (eager example)
- AOTAutograd: How we handle tracing the autograd engine in torch.compile
- Other things AOTAutograd does: functionalization
- Dynamo vs. AOTAutograd tracing: differences

Problem Statement
Automatic Differentation
PyTorch’s Autograd
AOTAutograd
Functionalization
Recap

Problem Statement

- Torch.compile should support training

Problem Statement
Automatic Differentation
PyTorch’s Autograd
AOTAutograd
Functionalization
Recap

Problem Statement

- Torch.compile should support training
- Training support requires capturing + compiling a backward graph
- **Autograd is implemented in C++ (PyTorch internals)**
    - **Dynamo cannot trace into PyTorch’s autograd code**

Problem Statement
Automatic Differentation
PyTorch’s Autograd
AOTAutograd
Functionalization
Recap

Background: Autodiff

Background: Autodiff

Steps when training a neural network:

1. Forward propagation
   - the network makes its best guess about the correct output. It runs the input data through each of its functions to make this guess
2. Loss function
   - Compute a scalar “loss”, dictating how far off the network was from the expected output
3. Backward propagation
   - Compute gradients which inform us of the direction in which we should move the network’s weights to minimize the loss
4. Optimizer step:
   - Update the network weights given the computed gradients

```
out = model(input)
loss = loss_fn(out, expected_out)
out.sum().backward()
optimizer.step()
```

Background: Autodiff

Gradient compute: derived automatically from the forward.

The user does not write python code corresponding to their backward.

```
out = model(input)
loss = loss_fn(out, expected_out)
out.sum().backward()
optimizer.step()
```

PyTorch: Autograd

Problem Statement
Automatic Differentation
PyTorch's Autograd
AOTAutograd
Functionalization
Recap

PyTorch: Autograd

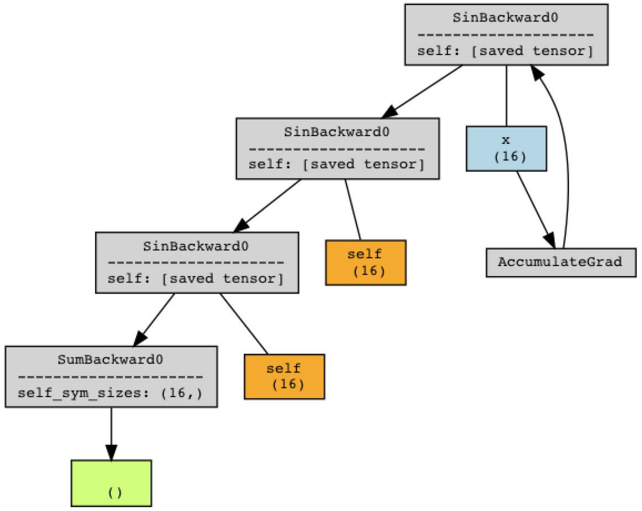
- PyTorch's autograd engine is tape-based
  - Calling operators on tensors will record their backward formulas into a "tape"
  - Every operator has a mapping to its derivative formula
    - sin(x) -> cos(x)
- Invoking .backward() will:
  - Execute each operator in the backward tape
  - Populate gradients into the .grad field

```
>>> x = torch.ones(4, requires_grad=True)
>>> out = x.sin()
>>> print(out.grad_fn)
<SinBackward0 object at 0x1051b0730>
>>> out.sum().backward()
>>> print(x.grad)
tensor([0.5403, 0.5403, 0.5403, 0.5403])
```

PyTorch: Autograd

We can visualize the backward graph created by the autograd engine!

```
import torch
from torchviz import make_dot

def f(x):
    return x.sin().sin().sin()

param = torch.randn(16, requires_grad=True)
out_expected = torch.zeros(16)
out = f(x)
loss = (out - out_expected) ** 2

make_dot(
    out.sum(),
    params={'x': param},
    show_attrs=True,
    show_saved=True
).render("bw_graph", format="png")
```



AOTAutograd

Back to torch.compile.

Training support: we want torch.compile to be able to compile both the forward and the backward

- The autograd engine logic is in C++
- No bytecode for dynamo to trace
- How do we get the backward graph?

Problem Statement
Automatic Differentation
PyTorch's Autograd
AOTAutograd
Functionalization
Recap

AOTAutograd

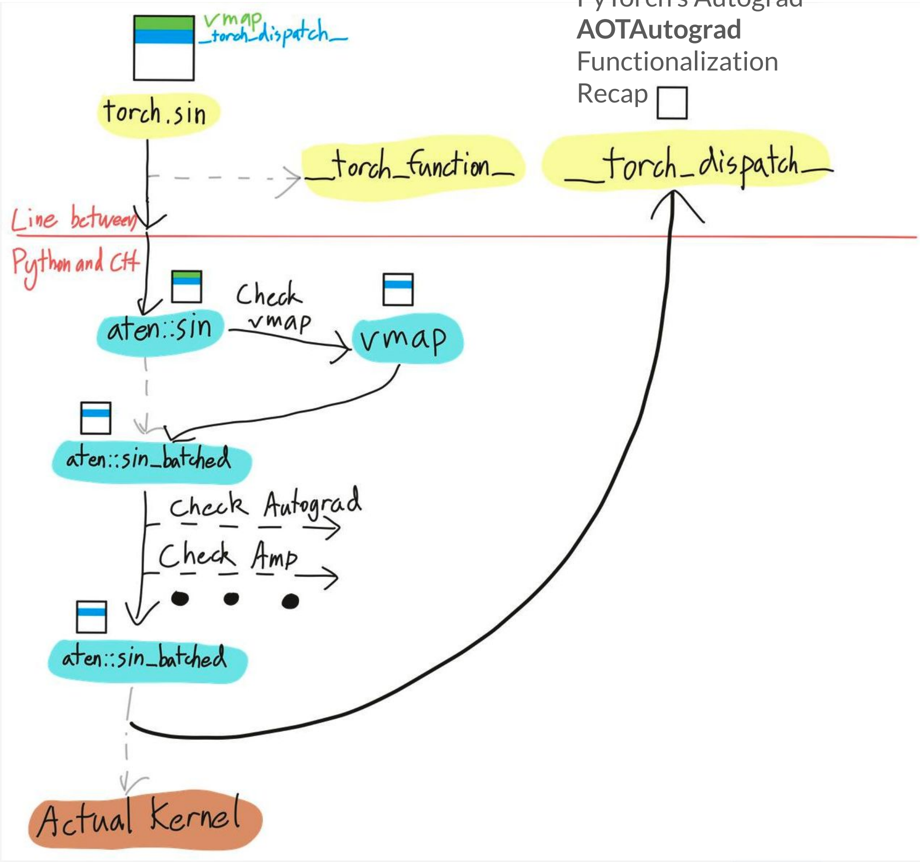
Back to torch.compile.

Training support: we want torch.compile to be able to compile both the forward and the backward

- The autograd engine logic is in C++
- No bytecode for dynamo to trace
- How do we get the backward graph?

__torch_dispatch__: a hook back into python, right before every operator is executed

When autograd executes its backward operations, intercept and record each operator into an FX graph



AOTAutograd: Example

User code

```
@torch.compile
def f(x):
    return x.sin().sin().sin()
```

AOTAutograd: Example

Step 1:

- Dynamo traces the python bytecode from the user
- Traces out an FX graph containing the user's torch operations

```
===== __compiled_fn_0 =====
<eval_with_key>.0 class GraphModule(torch.nn.Module):
    def forward(self, L_x_ : torch.Tensor):
        l_x_ = L_x_

        # File: /Users/hirsheybar/tmp.py:25, code: return x.sin().sin().sin()
        sin = l_x_.sin();  l_x_ = None
        sin_1 = sin.sin();  sin = None
        sin_2 = sin_1.sin();  sin_1 = None
        return (sin_2,)
```

1 graph output

User code

```
@torch.compile
def f(x):
    return x.sin().sin().sin()
```

AOTAutograd: Example

Step 2:

- AOTAutograd takes the “forward” graph from Dynamo
- Traces through the autograd engine
- Generates separate graphs for the forward and backward

User code

```
@torch.compile
def f(x):
    return x.sin().sin().sin()
```

AOTAutograd: Example

Step 2:

- AOTAutograd takes the "forward" graph from Dynamo
- Traces through the autograd engine
- Generates separate graphs for the forward and backward

Forward graph

2 graph outputs:

- The user output (result of sin)
- Saved activation, used in the backward pass

```
===== Forward graph 0 =====
<eval_with_key>.35 class GraphModule(torch.nn.Module):
    def forward(self, primals_1: "f32[16]"):
        sin: "f32[16]" = torch.ops.aten.sin.default(primals_1)
        sin_1: "f32[16]" = torch.ops.aten.sin.default(sin)
        sin_2: "f32[16]" = torch.ops.aten.sin.default(sin_1)
        return [sin_2, primals_1]
```

User code

```
@torch.compile
def f(x):
    return x.sin().sin().sin()
```

AOTAutograd: Example

Step 2:

- AOTAutograd takes the "forward" graph from Dynamo
- Traces through the autograd engine
- Generates separate graphs for the forward and backward

Backward graph

1 graph output:

- Gradient of output w.r.t. x

```
===== Backward graph 0 =====
<eval_with_key>.36 class GraphModule(torch.nn.Module):
    def forward(self, primals_1: "f32[16]", tangents_1: "f32[16]"):
        sin: "f32[16]" = torch.ops.aten.sin.default(primals_1)
        sin_1: "f32[16]" = torch.ops.aten.sin.default(sin)
        cos: "f32[16]" = torch.ops.aten.cos.default(sin_1)
        mul: "f32[16]" = torch.ops.aten.mul.Tensor(tangents_1, cos)
        cos_1: "f32[16]" = torch.ops.aten.cos.default(sin)
        mul_1: "f32[16]" = torch.ops.aten.mul.Tensor(mul, cos_1)
        cos_2: "f32[16]" = torch.ops.aten.cos.default(primals_1)
        mul_2: "f32[16]" = torch.ops.aten.mul.Tensor(mul_1, cos_2)
        return [mul_2]
```

User code

```
@torch.compile
def f(x):
    return x.sin().sin().sin()
```

What happens at runtime

User code

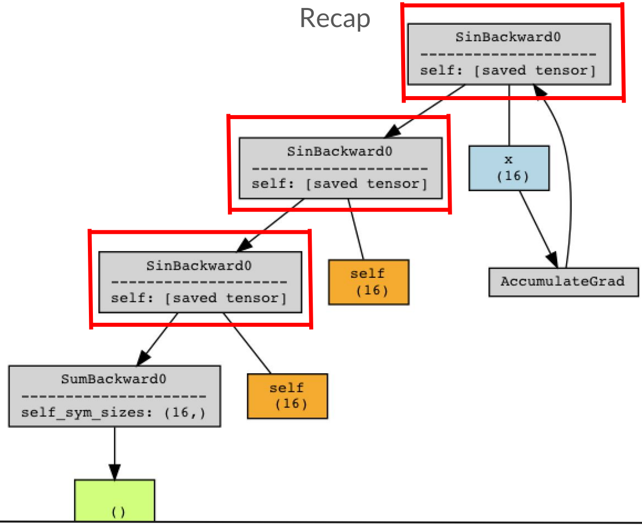
```
def f(x):
    return x.sin().sin().sin()

x = torch.randn(16, requires_grad=True)
out = f(x)
print(out.grad_fn)
```

Problem Statement
Automatic Differentiation
PyTorch's Autograd
AOTAutograd
Functionalization
Recap

What happens at runtime

In eager mode

- 3 ops in backward
- Every sin() saves its input for backward

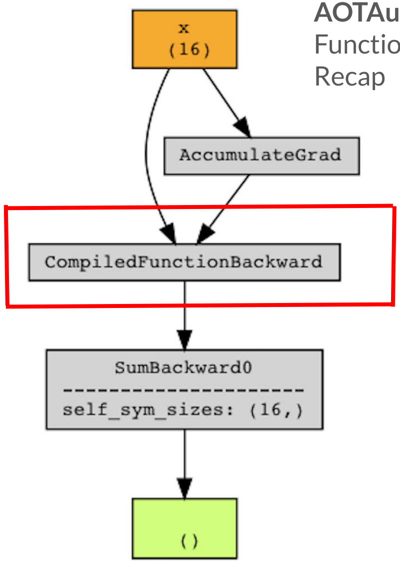


User code

```
def f(x):
    return x.sin().sin().sin()

x = torch.randn(16, requires_grad=True)
out = f(x)
print(out.grad_fn)
```

```
<SinBackward0 object at 0x150b22710>
```

What happens at runtime



With torch.compile

- 1 op in backward: "Compiled backward"
- No need to save all 3 tensors for backward

User code

```python
def f(x):
    return x.sin().sin().sin()

x = torch.randn(16, requires_grad=True)
out = f(x)
print(out.grad_fn)
```

```
<torch.autograd.function.CompiledFunctionBackward object at 0x154e885e0>
```

What else does AOTAutograd handle

What else does AOTAutograd handle

AOTAutograd traces "framework" code that lives in C++

- Motivating use case: autograd engine (for training support)
- But many other functionalities in PyTorch are implemented in C++ framework code
  - (for eager performance)
- AOTAutograd traces through these too
  - AMP (automatic mixed precision)
  - Functorch transforms (vmap/grad)
  - Tensor subclasses (user-land extension point)
  - Operator decompositions
  - **Functionalization (remove mutations from a program)**

Problem Statement
Automatic Differentation
PyTorch’s Autograd
AOTAutograd
Functionalization
Recap

Functionalization

Problem Statement
Automatic Differentation
PyTorch’s Autograd
AOTAutograd
Functionalization
Recap

Functionalization

- Another transform that lives in C++
- Removes mutations from a program
  - Compilers prefer a functional graph
- AOTAutograd traces through it too

```
@torch.compile(backend="aot_eager")
def f(x):
    a = x.add(1)
    a.add_(2)
    return a.add(3)

x = torch.ones(4, 4)
out = f(x)
```

Functionalization

- Another transform that lives in C++
- Removes mutations from a program
  - Compilers prefer a functional graph
- AOTAutograd traces through it too

Dynamo graph: has mutation

```
def forward(self, l_x_ : torch.Tensor):
    # File: /Users/hirsheybar/tmp2.py:5, code: a = x.add(1)
    a = l_x_.add(1)

    # File: /Users/hirsheybar/tmp2.py:6, code: a.add_(2)
    add_ = a.add_(2)

    # File: /Users/hirsheybar/tmp2.py:7, code: return a.add(3)
    add_1 = a.add(3)
    return (add_1,)
```

```
@torch.compile(backend="aot_eager")
def f(x):
    a = x.add(1)
    a.add_(2)
    return a.add(3)

x = torch.ones(4, 4)
out = f(x)
```

Problem Statement
Automatic Differentiation
PyTorch’s Autograd
AOTAutograd
Functionalization
Recap

Functionalization

- Another transform that lives in C++
- Removes mutations from a program
  - Compilers prefer a functional graph
- AOTAutograd traces through it too

AOTAutograd graph: functional

```
def forward(self, arg0_1: "f32[4, 4]"):
    # File: /Users/hirsheybar/tmp2.py:5, code: a = x.add(1)
    add: "f32[4, 4]" = torch.ops.aten.add.Tensor(arg0_1, 1)

    # File: /Users/hirsheybar/tmp2.py:6, code: a.add_(2)
    add_1: "f32[4, 4]" = torch.ops.aten.add.Tensor(add, 2)

    # File: /Users/hirsheybar/tmp2.py:7, code: return a.add(3)
    add_2: "f32[4, 4]" = torch.ops.aten.add.Tensor(add_1, 3)
    return (add_2,)
```

```
@torch.compile(backend="aot_eager")
def f(x):
    a = x.add(1)
    a.add_(2)
    return a.add(3)

x = torch.ones(4, 4)
out = f(x)
```

Back to the overview

Back to the overview

Dynamo:

- traces the user's python bytecode, puts all torch.* operators into a graph
- Compiling this graph is hard! Mutation, aliasing, backward is implicitly defined

AOTAutograd

- Generates a "simpler" graph, given the graph from dynamo
- Does so by tracing the PyTorch "framework" code that lives in C++

User code (good for Dynamo)	(PyTorch) Framework code (good for AOTAutograd)

• All in Python
• Can do arbitrarily crazy things
  ○ Lots of global state
  ○ 3rd party libs
• Mix of "tensor compute" and python side effects
	• Mostly in C++
	• Does not do arbitrarily crazy things (easy to trace)
	  ○ We control it (can tweak make it tracer-friendly)
	  ○ No 3rd party libs
	• 100% tensor compute