PyTorch Export

Sound Whole-Graph Capture for PyTorch

Avik Chaudhuri (PyTorch Compiler, AI at Meta)

☯

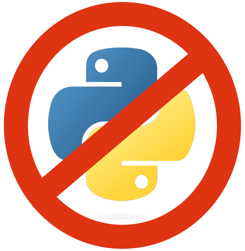
`torch.compile`	`torch.export`
just-in-time compiler	
needs Python runtime	
emits **backend-specific code**	
doesn’t need source changes	

`torch.compile`	`torch.export`
just-in-time compiler	**ahead-of-time frontend** compiler
needs Python runtime	**cuts dependency** on Python runtime
emits **backend-specific code**	emits **backend-agnostic IR**
doesn’t need source changes	**compile-time errors** possible

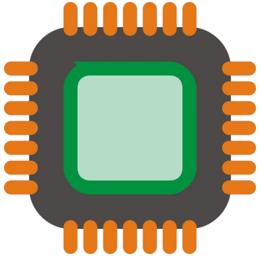
Why export?

AOT (instead of JIT) compilation is often…

- **necessary**
  - Python-free environments
    - e.g., on-device, serving



- **desirable**
  - backend-specific optimizations
    - e.g., custom kernels, special hardware
  - stability
    - e.g., online/recurring training



Graph breaks expected for JIT

```python
@torch.compile()
class M(torch.nn.Module):
  def forward(self, x):
    if x.sum() > 0:
      return x.sin()
    else:
      return x + 1

m = M()
```

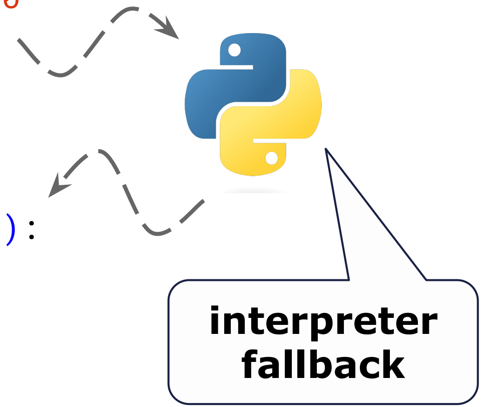
dynamic control flow

Graph breaks expected for JIT

```
@torch.compile()
class M(torch.nn.Module):
  def forward(self, x):
    if x.sum() > 0:
      return x.sin()
    else:
      return x + 1

m = M()
m(torch.randn(8))
```

```
def graph_0(x):
  sum = x.sum()
  gt = sum > 0
  return gt
```



interpreter fallback

```
def graph_1(x):
  add = x + 1
  return add
```

Graph breaks considered harmful for AOT

```
class M(torch.nn.Module):
  def forward(self, x):
    return torch.cond(
      x.sum() > 0,
      lambda x: x.sin(),
      lambda x: x + 1,
      (x,),
    )

m = M()
torch.export(m, (torch.randn(8),))
```

rewrite

```
def graph_0(x):
  sum = x.sum()
  gt = sum > 0

  def true_fn(x):
    sin = x.sin()
    return sin

  def false_fn(x):
    add = x + 1
    return add

  cond = torch.ops.higher_order.cond(
    gt, true_fn, false_fn, (x,)
  )
  return cond
```

Recompilations expected for JIT

```
@torch.compile()
class M(torch.nn.Module):
  def forward(self, x, y):
    if x.shape[0] <= 1024:
      return x + y
    else:
      return x

m = M()
```

static control flow

[Digression] ML compiler: shapes :: PL compiler: types!

Slogan: shapes are types!

complexity

Python types (e.g., zip)	**PyTorch shapes (e.g., matmul)**
`(List, List) -> List`	`(Tensor, Tensor) -> Tensor`

[Digression] ML compiler: shapes :: PL compiler: types!

Slogan: shapes are types!

complexity

Python types (e.g., zip)	**PyTorch shapes (e.g., matmul)**
`(List, List) -> List`	`(Tensor, Tensor) -> Tensor`
`(List[int], List[str]]) -> List[(int, str)]`	`(Tensor[4,8], Tensor[8,16]) -> Tensor[4,16]`

[Digression] ML compiler: shapes :: PL compiler: types!

Slogan: shapes are types!

complexity

Python types (e.g., zip)	**PyTorch shapes (e.g., matmul)**
`(List, List) -> List`	`(Tensor, Tensor) -> Tensor`
`(List[int], List[str]]) -> List[(int, str)]`	`(Tensor[4,8], Tensor[8,16]) -> Tensor[4,16]`
`(List[X], List[Y]) -> List[(X, Y)]`	`(Tensor[a,b], Tensor[b,c]) -> Tensor[a,c]`

Recompilations expected for JIT

```
@torch.compile()
class M(torch.nn.Module):
  def forward(self, x, y):
    if x.shape[0] <= 1024:
      return x + y
    else:
      return x

m = M()
```

static control flow

Recompilations expected for JIT

```python
@torch.compile()
class M(torch.nn.Module):
  def forward(self, x, y):
    if x.shape[0] <= 1024:
      return x + y
    else:
      return x

m = M()
m(torch.randn(512), torch.randn(512))
```

```python
def graph_0(x: f[512], y: f[512]):
  add: f[512] = x + y
  return add
```

static shapes

Recompilations expected for JIT

```
@torch.compile()
class M(torch.nn.Module):
  def forward(self, x, y):
    if x.shape[0] <= 1024:
      return x + y
    else:
      return x

m = M()
m(torch.randn(512), torch.randn(512))
m(torch.randn(1024), torch.randn(1024))
```

```
# s <= 1024
def graph_1(x: f[s], y: f[s]):
  add: f[s] = x + y
  return add
```

~~static shapes~~
dynamic shapes

Recompilations expected for JIT

```
@torch.compile()
class M(torch.nn.Module):
  def forward(self, x, y):
    if x.shape[0] <= 1024:
      return x + y
    else:
      return x

m = M()
m(torch.randn(512), torch.randn(512))
m(torch.randn(1024), torch.randn(1024))
m(torch.randn(2048), torch.randn(1024))
```

```
# s > 1024
def graph_2(x: f[s], y):
  return x
```

~~static shapes~~

dynamic shapes

(new path)

Recompilations expected for JIT

```
@torch.compile()
class M(torch.nn.Module):
  def forward(self, x, y):
    if x.shape[0] <= 1024:
      return x + y
    else:
      return x

m = M()
m(torch.randn(512), torch.randn(512))
m(torch.randn(1024), torch.randn(1024))
m(torch.randn(2048), torch.randn(1024))
m(torch.randn(1024), torch.randn(2048))
```

```
# s <= 1024
def graph_1(x: f[s], y: f[s?]):
  add: f[s?] = x + y
  return add
```

~~static shapes~~
~~dynamic shapes~~
~~(new path)~~
run-time exception

~~torch.compile (just-in-time)~~ torch.export (ahead-of-time)

Recompilations considered harmful for AOT

```
class M(torch.nn.Module):
  def forward(self, x, y):
    if x.shape[0] <= 1024:
      return x + y
    else:
      return x
```

```
def graph_0(x: f[512], y: f[512]):
  add: f[512] = x + y
  return add
```

static shapes

```
m = torch.export(M(), (torch.randn(512), torch.randn(512)).module()
```

~~torch.compile (just-in-time)~~ torch.export (ahead-of-time)

Recompilations considered harmful for AOT

```
class M(torch.nn.Module):
  def forward(self, x, y):
    if x.shape[0] <= 1024:
      return x + y
    else:
      return x
```

```
def graph_0(x: f[512], y: f[512]):
  add: f[512] = x + y
  return add
```

static shapes

```
m = torch.export(M(), (torch.randn(512), torch.randn(512)).module()
```

```
m(torch.randn(1024), torch.randn(1024))
```

assertion failure
(shapes not dynamic)

~~torch.compile (just-in-time)~~ torch.export (ahead-of-time)

Recompilations considered harmful for AOT

```
class M(torch.nn.Module):
  def forward(self, x, y):
    if x.shape[0] <= 1024:
      return x + y
    else:
      return x
```

```
# s <= 1024
def graph_1(x: f[s], y: f[s]):
  add: f[s] = x + y
  return add
```

dynamic shapes

```
args = (torch.randn(512), torch.randn(512))
s = torch.export.Dim(“s”, max=1024)
m = torch.export(M(), args, dynamic_shapes=((s), (s))).module()

m(torch.randn(1024), torch.randn(1024))
```

ok

Recompilations considered harmful for AOT

```
class M(torch.nn.Module):
  def forward(self, x, y):
    if x.shape[0] <= 1024:
      return x + y
    else:
      return x
```

```
# s <= 1024
def graph_1(x: f[s], y: f[s]):
  add: f[s] = x + y
  return add
```

dynamic shapes

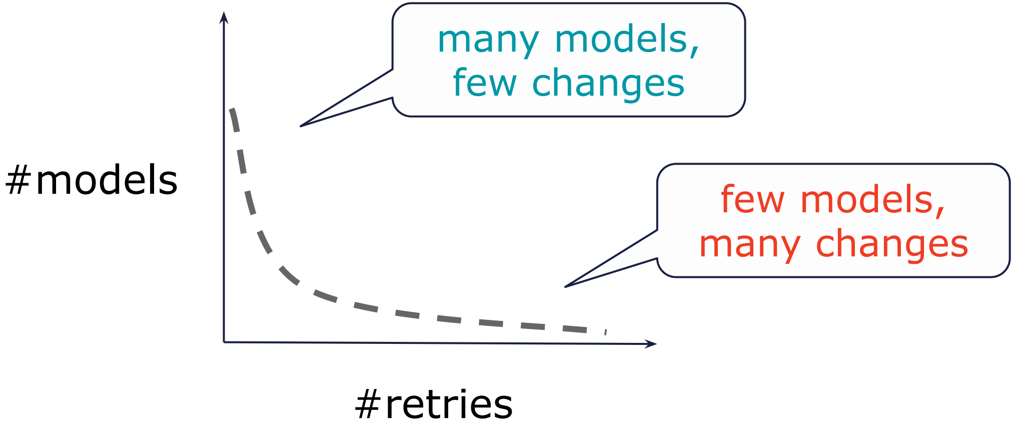
```
args = (torch.randn(512), torch.randn(512))
s = torch.export.Dim(“s”, max=1024)
m = torch.export(M(), args, dynamic_shapes=((s), (s))).module()

m(torch.randn(1024), torch.randn(1024))
m(torch.randn(2048), torch.randn(1024))
m(torch.randn(1024), torch.randn(2048))
```

ok

assertion failure (shape constraints violated)

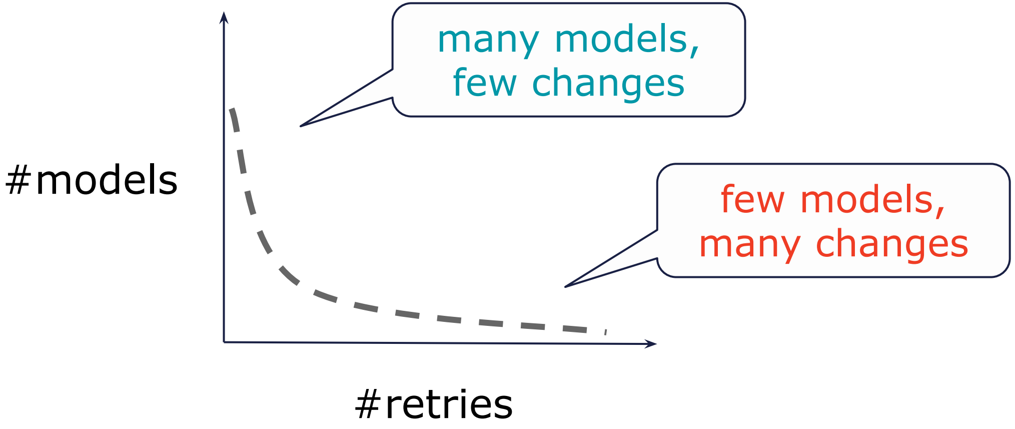
Data-driven UX



Data-driven UX

Focus

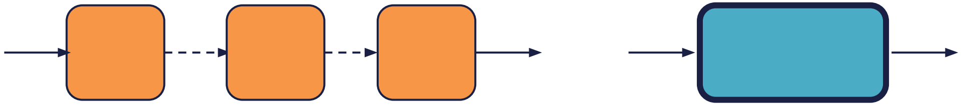
- **Simplicity**
  - predictable behaviors
- **Actionable errors**
  - source locations
  - suggested fixes
- **Expressivity**
  - control flow, state
  - shape constraints



Guarantees of IR

Syntactic guarantees

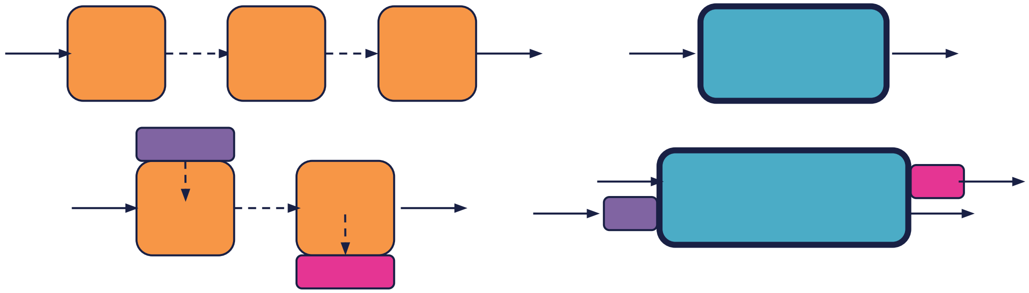
- **Inlining**
  - flattened graph



Guarantees of IR

Syntactic guarantees

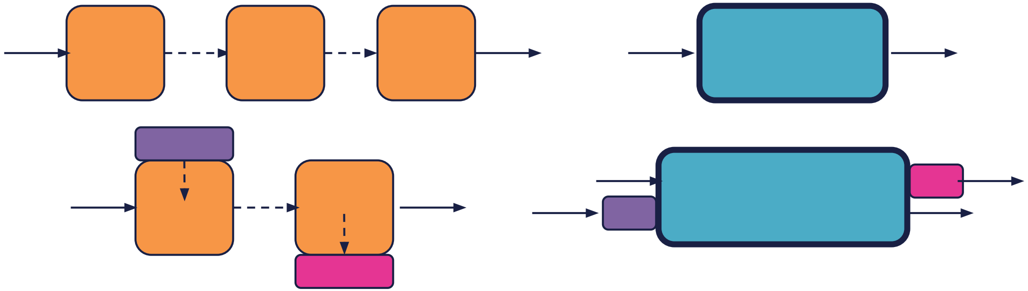
- **Inlining**
  - flattened graph
- **Functionalization**
  - lifted inputs/outputs



Guarantees of IR

Syntactic guarantees

- **Inlining**
  - flattened graph
- **Functionalization**
  - lifted inputs/outputs
- **Metadata**
  - source map (unlift+unflatten)
  - shape constraints (soundness)



Guarantees of IR

Syntactic guarantees

- **Inlining**
  - flattened graph
- **Functionalization**
  - lifted inputs/outputs
- **Metadata**
  - source map
  - shape constraints

Semantic guarantees

- **Behavioral correctness**
  - assume / guarantee

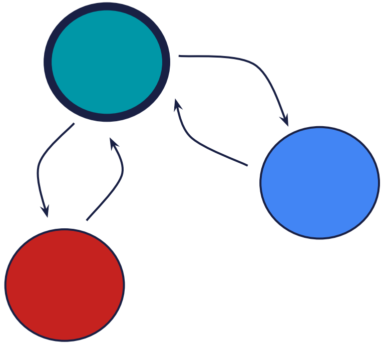
Guarantees of IR

Syntactic guarantees

- **Inlining**
  - flattened graph
- **Functionalization**
  - lifted inputs/outputs
- **Metadata**
  - source map
  - shape constraints

Semantic guarantees

- **Behavioral correctness**
  - assume / guarantee
- **Round-tripping identities**
  - serialize / deserialize
  - unlift+unflatten / re-export



Customization [operator decomposition]

```
# “full” aten opset
exported = ...

# “core” aten opset (standard for maximum coverage)
exported_core = exported.run_decompositions()

# plug-in custom kernel for op
exported_custom = exported.run_decompositions(
  decomp_table=...  # default except op
)
```

e.g., convolution

APIs for backends

Transformation [graph optimization]

```
# functionalized, linear FX graph
exported = ...

# sequence of arbitrary FX transforms
lowered = optimization_passes(exported)

# execute!
... = backend(lowered)
```

ONNX
Torch-XLA
TensorRT
ExecuTorch
...
AOTInductor