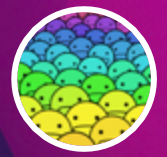
# AI编译器-系列之PyTorch

# TorchFX & Lazy Tensor



ZOMI

# Talk Overview

1. **PyTroch 2.0 新特性**
   - 2.0 新特性回顾
   - PyTorch 2.0 安装与新特性使用
   - PyTorch 2.0 对厂商的启发和思考
2. **TorchDynamo 解读**
   - TorchDynamo 特性
   - TorchDynamo 实现方案
3. **AOTAutograd 解读**
   - AOTAutograd 效果
   - AOTAutograd 实现方案
4. **TorchInductor 新特性**
   - Triton 使用解读
   - Triton 深度剖析

# Talk Overview

## 1. TorchDynamo 解读

- PyTorch 获取计算图的方式
- PyTorch JIT Script 原理
- PyTorch FX 原理
- PyTorch Lazy Tensor 原理
- TorchDynamo 原理

# Torch FX 原理

# Design Principles

- Prefer making program capture and transformation easy for typical models at the cost of working for all possible programs. Avoid complexity to support longtail, esoteric use cases.
- Work with tools and concepts that ML practitioners are already familiar with such as Python data structures and the publicly documented operators in PyTorch.
- Make the process of program capture highly configurable so users can implement their own solutions for long-tail uses. Allowing users to make one-off configurations is simpler than handling the general case.

# TORCH.FX NODE SEMANTICS

- torch.fx represents programs in a DAG-based IR, which contains a linear series of Node objects representing operations. Nodes have a string opcode, describing what type of operation the Node represents. Nodes have an associated target, which is the call target for call nodes (call_module, call_function, and call_method).

### A.1 Opcode Meanings

| Opcode | Meaning |
|---|---|
| `placeholder` | Function Input |
| `call_method` | Call method on `args[0]` |
| `call_module` | Call module specified by `target` |
| `call_function` | Call function specified by `target` |
| `get_attr` | Retrieve attribute specified by `target` |
| `output` | Return statement; return `args[0]` |

### A.2 `args/kwargs` Behavior

| Opcode | `args/kwargs` Behavior |
|---|---|
| `placeholder` | Empty or `args[0]` = default value |
| `call_method` | Python calling convention; `args[0]` is self |
| `call_module` | Python calling convention; `target` is self |
| `call_function` | Python calling convention; `target` is self |
| `get_attr` | Empty |
| `output` | `args[0]` is the return value |

# TORCH.FX IR

1. placeholder represents a function input. The name attribute specifies the name this value will take on. target is similarly the name of the argument. args holds either: 1) nothing, or 2) a single argument denoting the default parameter of the function input. kwargs is don't-care. Placeholders correspond to the function parameters (e.g. x) in the graph printout.
2. get_attr retrieves a parameter from the module hierarchy. name is similarly the name the result of the fetch is assigned to. target is the fully-qualified name of the parameter's position in the module hierarchy. args and kwargs are don't-care
3. call_function applies a free function to some values. name is similarly the name of the value to assign to. target is the function to be applied. args and kwargs represent the arguments to the function, following the Python calling convention
4. call_module applies a module in the module hierarchy's forward() method to given arguments. name is as previous. target is the fully-qualified name of the module in the module hierarchy to call. args and kwargs represent the arguments to invoke the module on, including the self argument.
5. call_method calls a method on a value. name is as similar. target is the string name of the method to apply to the self argument. args and kwargs represent the arguments to invoke the module on, including the self argument
6. output contains the output of the traced function in its args[0] attribute. This corresponds to the "return" statement in the Graph printout.

# TORCH.FX Example

```python
import torch
from torch.fx import symbolic_trace, GraphModule

def foo(x, y):
    a = torch.sin(x)
    b = torch.cos(x)
    return a + b

traced = symbolic_trace(foo)
print(traced.graph)
```

```
graph():
    %x : [#users=2] = placeholder[target=x]
    %y : [#users=0] = placeholder[target=y]
    %sin : [#users=1] = call_function[target=torch.sin](args = (%x,), kwargs = {})
    %cos : [#users=1] = call_function[target=torch.cos](args = (%x,), kwargs = {})
    %add : [#users=1] = call_function[target=operator.add](args = (%sin, %cos), kwargs = {})
    return add
```

# TORCH.FX Pros and Cons

**Pros**

- 能够对正向计算图进行操作，如批量增加、修改某个操作，增加统计操作，如量化；
- 是一个Python-to-Python的符号Trace方式，方便学习和进行源码操作；
- IR 表示缩减到 6 个操作，操作简单和 IR 易学；

**Cons：**

- 静态图只能表达正向图，不能够处理反向图和表示带有控制流的图，使用场景有限；
- Python-to-Python的Trace操作，继承了Trace Base转静态图的缺点；
- 需要用户感知替换，只能处理量化、融合算子等有限情况；

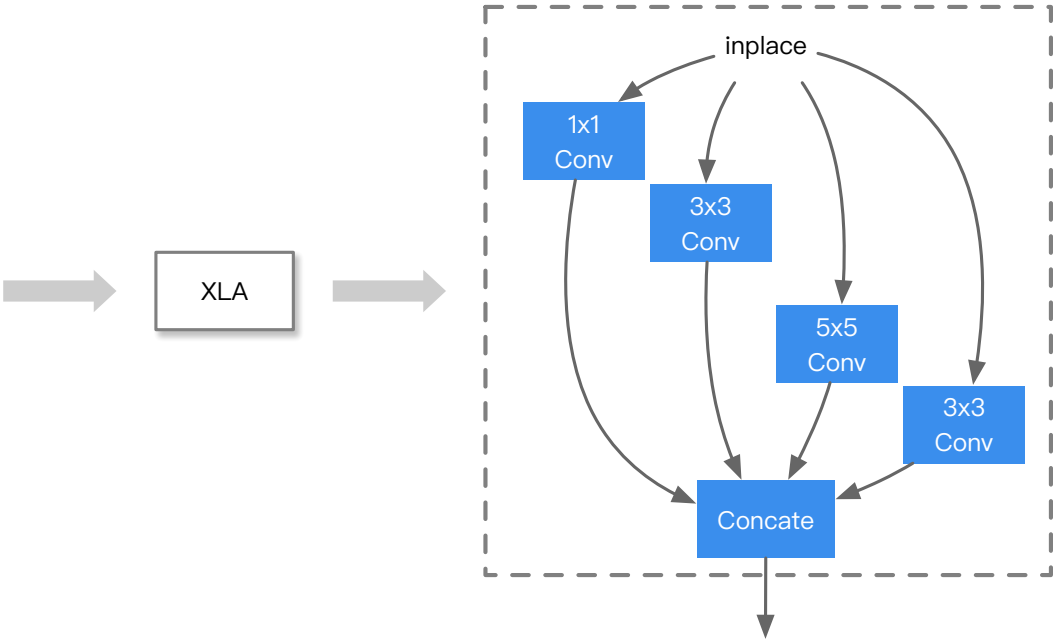
# Lazy Tensor 原理

# LazyTensor Principle

- Any operation performed on a PyTorch tensor is by default dispatched as a kernel or a composition of kernels to the underlying hardware. These kernels are executed asynchronously on the underlying hardware. The program execution is not blocked until the value of a tensor is fetched. This approach scales extremely well with massively parallel programmed hardware such as GPUs.

```python
import torch
import torch_xla.core.xla_model as xm


dev = xm.xla_device()

x1 = torch.conv2d((input, 3))
x2 = torch.conv2d((3, 8))
x3 = torch.conv2d((3, 16))
x4 = torch.conv2d((3, 24))
out = torch.concat(x1 , x2, x3, x4)

```

Python Code




异步缓存+编译

计算图执行

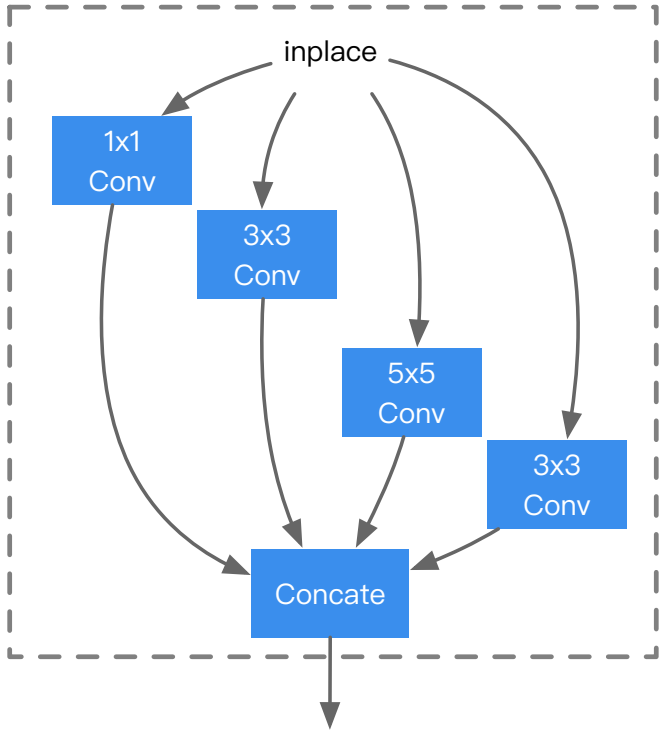
# LazyTensor Principle

```python
import torch
import torch_xla.core.xla_model as xm


dev = xm.xla_device()

x1 = torch.conv2d((input, 3))
x2 = torch.conv2d((3, 8))
x3 = torch.conv2d((3, 16))
x4 = torch.conv2d((3, 24))
out = torch.concat(x1 , x2, x3, x4)

```

Python Code

XLA

异步缓存+编译




计算图执行

# LazyTensor Demo

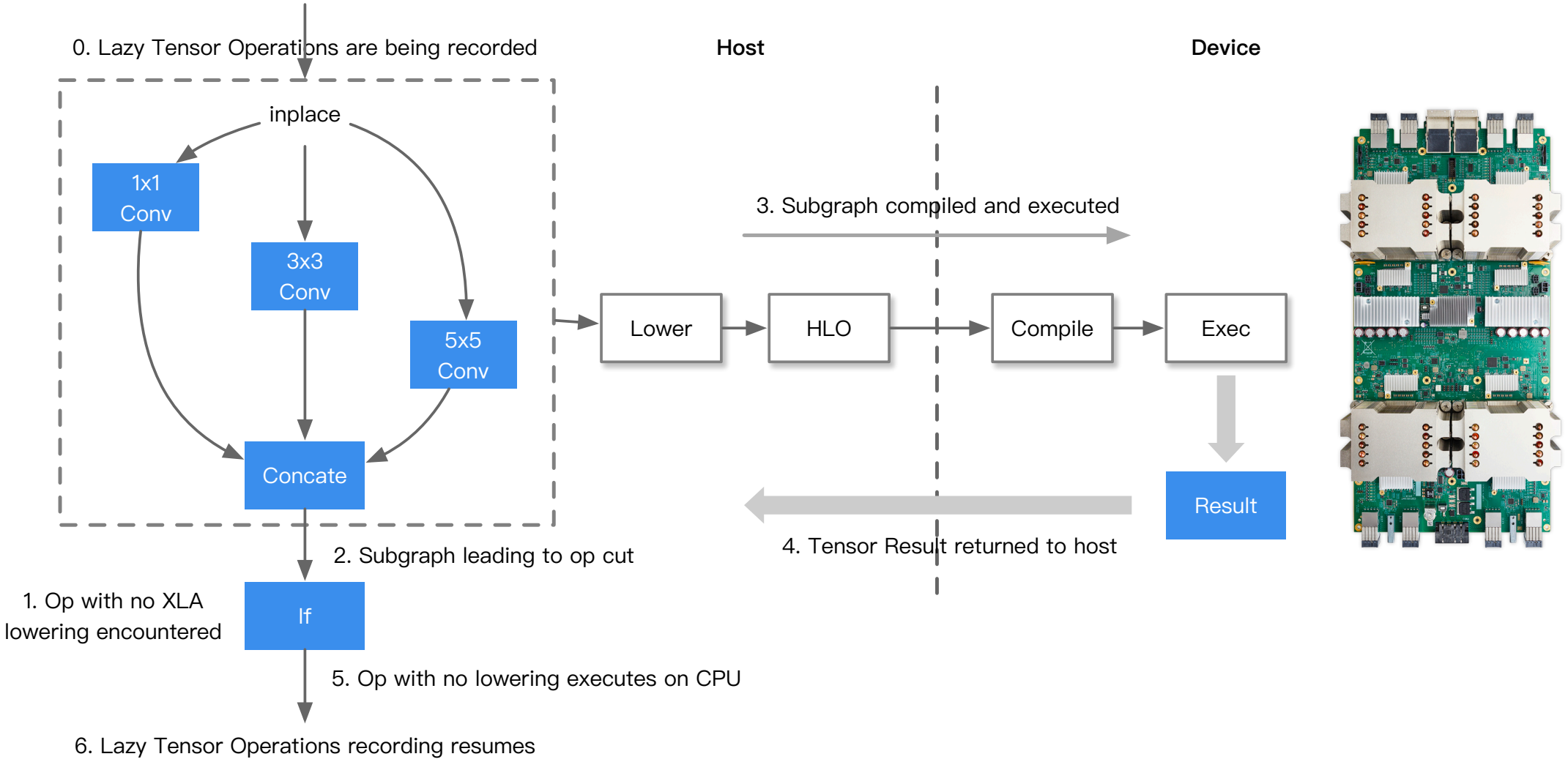
- The starting point of a LazyTensor system is a custom tensor type. In PyTorch/XLA, this type is called XLA tensor. In contrast to PyTorch's native tensor type, operations performed on XLA tensors are recorded into an IR graph. This barrier can either be a mark step() api call or any other event which forces the execution of the graph recorded so far.

```
1   import torch
2   import torch_xla
3   import torch_xla.core.xla_model as xm
4
5   dev = xm.xla_device()
6
7   x1 = torch.rand((3, 3)).to(dev)
8   x2 = torch.rand((3, 8)).to(dev)
9
10  y1 = torch.einsum('bs,st->bt', x1, x2)
11  print(torch_xla._XLAC._get_xla_tensors_text([y1]))
```

# Graph Compilation and Execution and LazyTensor Barrier

1. The first scenarios is the explicit call of mark_step() api as shown in the preceding example. mark_step() is also called implicitly at every step when you wrap your dataloader with MpDeviceLoader.
2. The second scenario where a barrier is introduced is when PyTorch/XLA finds an op with no mapping (lowering) to equivalent XLA HLO ops. PyTorch has 2000+ operations， some of these operations do not have corresponding lowering in XLA.
3. The third and final scenario which results in a LazyTensor barrier is when there is a control structure/statement or another method which requires the value of a tensor. This statement would at the minimum cause the execution of the computation graph leading to the tensor or cause compilation and execution of both.

# Graph Compilation and Execution and LazyTensor Barrier



# LazyTensor Pros and Cons

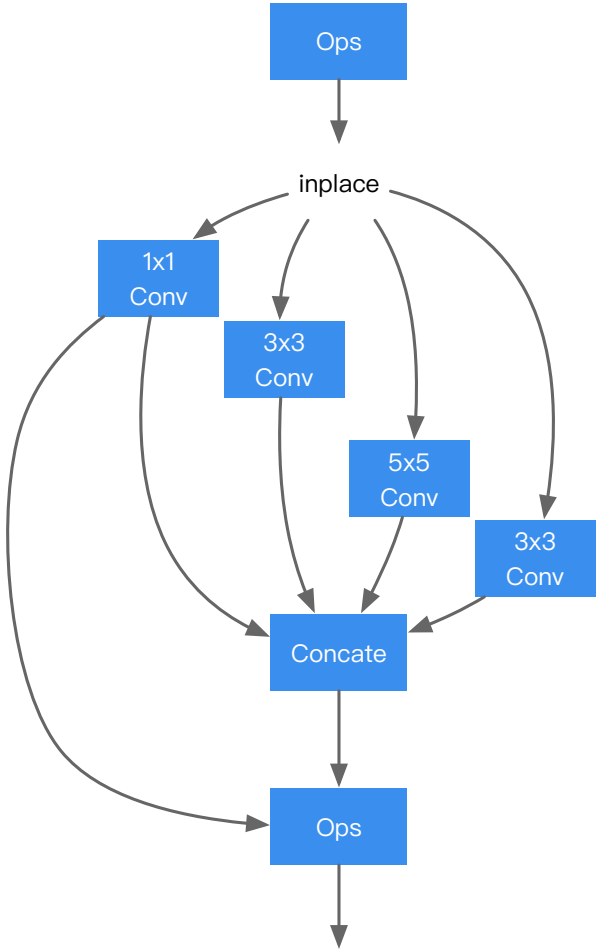
**Pros：**

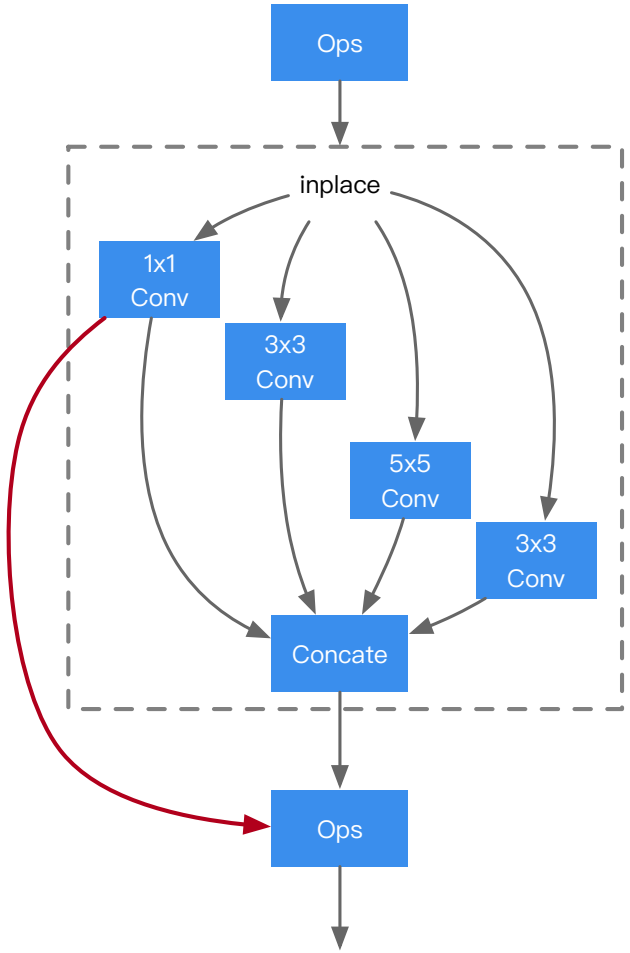
- LazyTensor在一定程度上能够带来优化收益
- 语法限制少，理论上只要找到合适的tensor操作序列都可以进行编译成子图

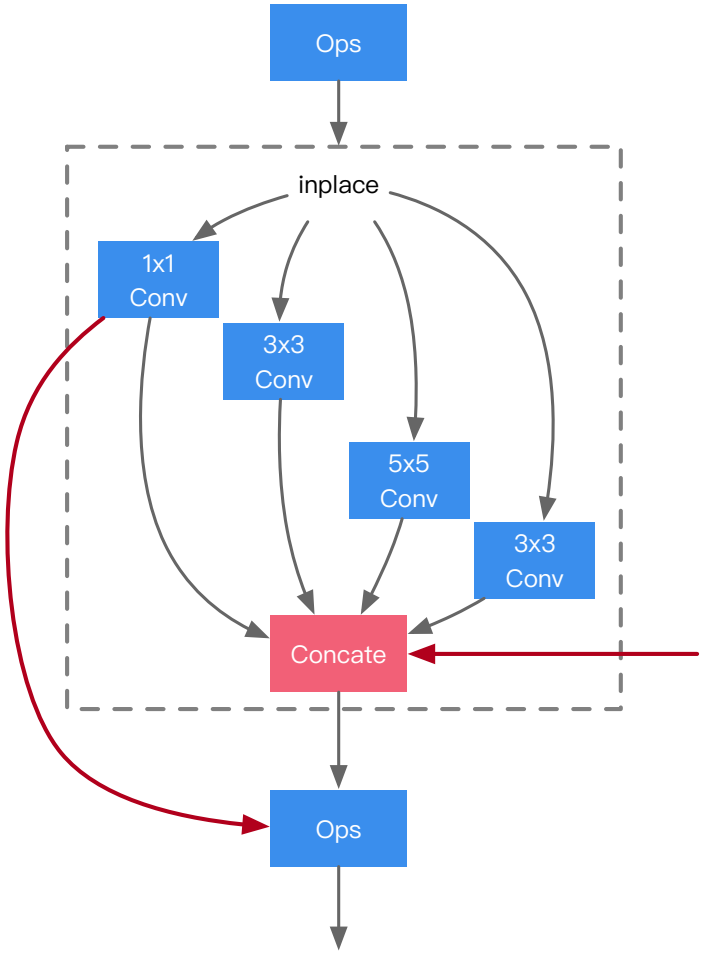
**Cons：**

- JIT 编译开销大，遇到特殊网络模型会来回重复异步调度（host-devices）。
- 动态情况下，如动态 shape 和控制流，每次变化都会触发新的编译，导致性能退化严重。
- 因为 fusion kernel 执行的原子特性，破坏了操作之间并行执行的可能性
- 影响 PyTorch eager mode 下 kernel 下发和执行可以重叠的异步计算，部分情况会增加执行开销

# LazyTensor Pros and Cons



# Comparation

| 特性 | 内容 | 适用场景 | 技术方式 |
| --- | --- | --- | --- |
| TorchScript | TorchScript 的主要用途是进行模型部署，需要记录生成一个便于推理优化的 IR，对计算图的编辑通常都是面向性能提升等等，不会给模型本身添加新的功能。 | 推理和部署静态图优化 | AST + Trace |
| Torch FX | FX 的主要用途是进行python->python的翻译，它的 IR 中节点类型更简单，比如函数调用、属性提取等等，这样的 IR 学习成本更低更容易编辑。使用 FX 来编辑图通常是为了实现某种特定功能，比如给模型插入量化节点等，避免手动编辑网络造成的重复劳动。 | 修改正向图 | Trace |
| LazyTensor | 采用 tensor 不被用户观察那么就可以缓存计算指令的方式，隐式构图+优化，用户不用标记构图的代码范围，对用户的代码限制小，是一种能优化就优化、不能优化也能执行的思路，偏向于易用性 | 针对NPU子图编译优化 | Trace |
| TorchDynamo | TorchDynamo 是 PyTorch 新实验的 JIT 编译接口，支持使用 Python 在运行时修改动态执行逻辑，修改的时机是 CPython 的 ByteCode 执行前。思想类似 DynamoRIO 项目，DynamoRIO 可以动态的修改 x86 机器码。 | 动静统一 + 编译优化 | Python 解析修改 |