Clad as a First-Class Citizen in LibTorch

A compiler-driven gradient path for pure C++ ML workflows

C++

Kacent Huang

2026-05-20

Compiler Research Team / GSoC 2026 draft

fogsong@gmail.com

About Me

- Second-year **undergraduate student** at **Nanjing University**.
- Working on a Clad-powered gradient engine for LibTorch through GSoC 2026.
- Interested in C++, compilers, programming languages, and machine learning.
- Experienced in C++ development and systems programming.

github: fogsong233

email: fogsong233@gmail.com

Motivation

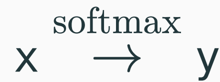
Motivation

- ROOT and HEP workflows increasingly use LibTorch for model inference and training in pure C++.
- Recent Clad work suggests compiler-generated gradients can be competitive with PyTorch autograd on CPU.
- This project explores whether Clad can become a practical gradient backend for selected LibTorch workloads.

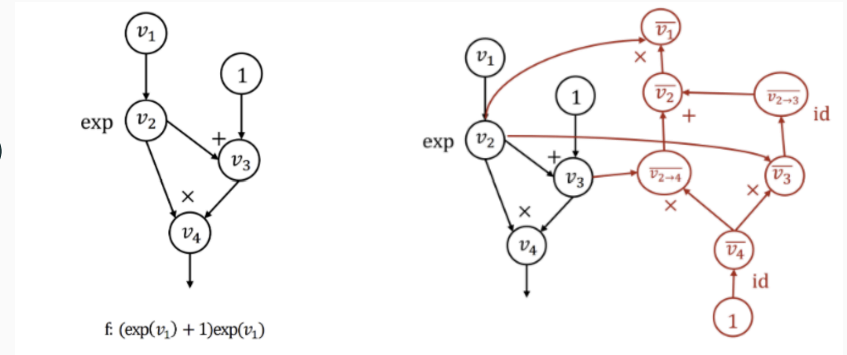
Problem Statement

Single Operator

```
auto z = x.softmax(-1);
```



`result.grad()` $\longrightarrow$



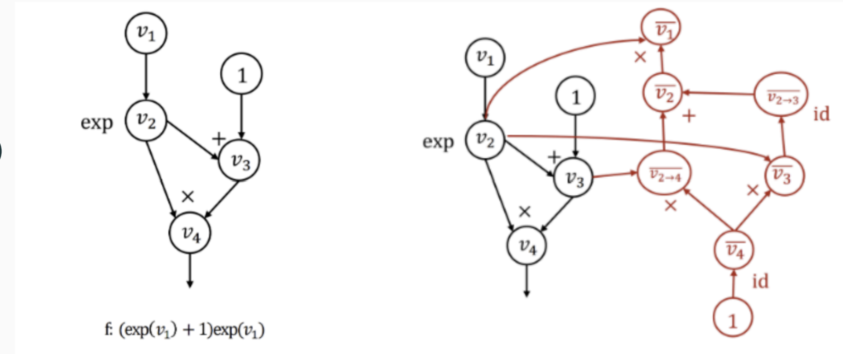
Choice 1: replace basic operator gradients with Clad.

Single Operator

```
auto z = x.softmax(-1);
```

$$\texttt{x} \xrightarrow{\text{softmax}} \texttt{y}$$

$\xrightarrow{\texttt{result.grad()}}$



Choice 1: replace basic operator gradients with Clad.

This is not very compelling by itself, because LibTorch already has a mature operator-gradient system.

$$\frac{\mathrm{d}L}{\mathrm{d}x} = \text{grad}\ \frac{\mathrm{d}\sqrt{x}}{\mathrm{d}x} = \frac{\text{grad}}{2\sqrt{x}} \quad \to$$

```
- name: sqrt(Tensor self) -> Tensor
  self: grad / (2 * result.conj())
  result: auto_element_wise
```

PyTorch records formulas like this to generate gradients for individual operators.

Choice 2: compile the dynamic graph

PyTorch: `@torch.compile()`

PyTorch: `@torch.compile()` captures Python-level execution and specializes the observed graph.

LibTorch(C++): No equivalent first-class workflow; eager mode is recommended

PyTorch: `@torch.compile()` captures Python-level execution and specializes the observed graph.

LibTorch(C++): **Use Clad to explore this missing path**

But the graph is still dynamic

In LibTorch eager mode, the graph is produced during execution:

```
auto my_graph(torch::Tensor x) -> torch::Tensor {
  auto y = (x * x).sum();
  return y;
}
```

But the graph is still dynamic

In LibTorch eager mode, the graph is produced during execution:

```
auto my_graph(torch::Tensor x) -> torch::Tensor {
  auto y = (x * x).sum();
  return y;
}
```

At runtime, this execution gives us an observed graph instance:

`x` $\rightarrow$ `aten::mul` $\rightarrow$ `aten::sum` $\rightarrow$ `y`

But the graph is still dynamic

In LibTorch eager mode, the graph is produced during execution:

```
auto my_graph(torch::Tensor x) -> torch::Tensor {
  auto y = (x * x).sum();
  return y;
}
```

At runtime, this execution gives us an observed graph instance:

`x` → `aten::mul` → `aten::sum` → `y`

Then we can specialize the observed graph by:

- operator sequence
- tensor dtype
- tensor shape
- layout / stride assumptions

But the graph is still dynamic

In LibTorch eager mode, the graph is produced during execution:

```
auto my_graph(torch::Tensor x) -> torch::Tensor {
  auto y = (x * x).sum();
  return y;
}
```

At runtime, this execution gives us an observed graph instance:

`x` → `aten::mul` → `aten::sum` → `y`

Then we can specialize the observed graph by:

- operator sequence
- tensor dtype
- tensor shape
- layout / stride assumptions

This is different from replacing a single operator gradient.

In LibTorch eager mode, the graph is produced during execution:

```
auto my_graph(torch::Tensor x) -> torch::Tensor {
  auto y = (x * x).sum();
  return y;
}
```

At runtime, this execution gives us an observed graph instance:

`x` → `aten::mul` → `aten::sum` → `y`

Then we can specialize the observed graph by:

- operator sequence
- tensor dtype
- tensor shape
- layout / stride assumptions

This is different from replacing a single operator gradient.

Clad is used as a backend for an observed graph instance, not as a replacement for each `aten::` operator rule.

User-facing code should still look like LibTorch:

```
auto my_graph(torch::Tensor x) -> torch::Tensor {
  return (x * x).sum();
}

auto compiled = cladtorch::compile(my_graph, example_inputs);
auto y = compiled.forward(inputs);
auto dx = compiled.backward(torch::ones_like(y));
```

User-facing code should still look like LibTorch:

```
auto my_graph(torch::Tensor x) -> torch::Tensor {
  return (x * x).sum();
}

auto compiled = cladtorch::compile(my_graph, example_inputs);
auto y = compiled.forward(inputs);
auto dx = compiled.backward(torch::ones_like(y));
```

The implementation can start with a tiny supported graph.

User-facing code should still look like LibTorch:

```
auto my_graph(torch::Tensor x) -> torch::Tensor {
  return (x * x).sum();
}

auto compiled = cladtorch::compile(my_graph, example_inputs);
auto y = compiled.forward(inputs);
auto dx = compiled.backward(torch::ones_like(y));
```

The implementation can start with a tiny supported graph.

To keep the first prototype realistic, we start with simple assumptions:

- CPU only
- contiguous float tensors
- a small selected operator set
- simple graph compositions first

Future vision

- Reuse PyTorch's derivative formulas to understand supported operator gradients.
- Lower observed graph instances into Clad-friendly C++ programs.
- Expand from tiny graphs to more realistic ML blocks.
- Fall back to LibTorch autograd when the graph is unsupported.

Project Goal

- Build a proof-of-concept for `torch.compile`-like graph capture in LibTorch C++.
- Use Clad to generate gradients for supported observed graph instances.
- Avoid changing basic ATen operator derivative rules.
- Keep LibTorch as the tensor runtime, while Clad acts as the graph-gradient compiler backend.

Thank You

Thank You

- Questions and feedback are welcome.