Abstract

Functionalization is a way to remove mutations from arbitrary PyTorch programs sent to downstream compilers.

The PyTorch 2.0 stack is all about capturing graphs of PyTorch operations and sending them off to a compiler to get better performance.

PyTorch programs can mutate and alias state, making them unfriendly to compilers.

Functionalization is a technique to take a program full of PyTorch operators, including mutable and aliasing operators, and remove all mutations from the program while preserving semantics.

Example Downstream Consumers

There are many compilers and optimizations that would prefer to operate on a functional graph.

- Backend-specific
- The XLA compiler accepts a graph of HLO IR as an input, which is entirely functional.
- Backend-agnostic:
  - AOTAutograd (in PyTorch core) re-orders operations between the forward backward to more optimally perform gradient checkpointing.

Algorithm

Aliasing refers to the case when two tensors (or objects) point to the same data.

Removing mutations when there is no aliasing is relatively simple:
For each mutation in the graph (e.g., `x.add(1)`):
1. Replace with functional equivalent: `x' = x.add(1)`
2. Replace all later usages of `x` with `x'`

Removing mutations from a program requires more work when aliasing is allowed:
For each mutation in the graph (e.g., `x.add(1)`):
1. Replace with functional equivalent: `x' = x.add(1)`
2. Replace all later usages of `x` with `x'`
3. Regenerate every alias of `x` (call it `v`) from `x'`: `v' = x'.view(...)`
4. Replace every later usage of an alias `v` with their new value `v'`

Acknowledgements

We’d like to thank the PyTorch/XLA team. The core algorithm underlying functionalization was first pioneered by PyTorch/XLA, as part of translating PyTorch into HLO IR.