Pytorch

Automatic Differentiation In Pytorch
Background and History

- Deep learning framework developed by Facebook
- Released in 2016
- Build upon previous frameworks like Chainer, Lua Torch, and HIPS
- Originally a Python library but has been moved to C++/C
What Does Pytorch Provide?

- Port of Torch framework into Python
- Support for GPU acceleration
- Integration with Numpy
- Automatically generated computational graphs
- Most importantly automatic differentiation
Automatic Differentiation

- Unlike symbolic differentiation, not interested in closed form solutions or a formula but in computing derivative at a point.
- Pytorch’s design decisions for its differentiation library are what set it apart from Tensorflow and inspired the changes in Tensorflow 2.0.
Pytorch
Autodifferentiation Key Features
Dynamic Execution

• Define-by-run
• No static graph structure specified; no sublanguage needed to create models
• Computational graph created as execution progresses
• Control flow remains normal and allows use of all Python features
Eager Execution

- Computations are completed as the program comes across them
- Information needed to compute gradients is stored
- Opposed to lazy execution where entire graph is created and then the full pass is computed
Pytorch
Autodifferentiation Novel Implementations
• **In-Place Operations:**
  Tensor operations can be performed without creating a copy. Can be useful for memory efficiency but pose a danger for automatic differentiation.

• **Tape:**
  Most deep learning frameworks make use of a tape (Wengert list) to keep track of the order of computations. Pytorch gets rid of this tape to allow for mixing graphs.

• **Speed Supported by C++:**
  Despite starting life as a Python library, Pytorch has evolved to move the core code into C++ to avoid interpreter overhead. Performance gains from Pytorch are largely from highly optimized C++ code being exposed to a Python interface.
In Place Operations

- In Place Operation: Any operation performed on a tensor where a copy is not needed, creates aliases of data
- Example: `y.add_(3)`
- Advantages: No memory copies needed, can save GPU memory
- Disadvantages: Can cause invalidation of data, nontrivial aliasing, had to implement properly
  - `X= y[:2]`
  - Creates an alias to the data in `y`, if an inplace operation is performed on `x` gradient would potentially be invalidated
  - Pytorch Solution: disable gradient operations in cases where aliases create problems rather than update all gradients and degrade performance
C++ Motivation

- Performance: Interpreter incurs high overhead and lower performance
- 1.7us tensor operation rather than 3.4us
- Throughput: GIL limits parallelism in Python, need multiple threads to saturate multiple GPUs
Pytorch Code Example

Colab Notebook
Pytorch Interface Features
Features

- **Hooks**: allows users to inspect gradients while PyTorch is running by registering a callback on them

  ```python
  x.register_hook(lamda grad: print(grad))
  ```

- **Extensibility**: Users are not limited to functions implemented by PyTorch, custom functions can be added by defining a forward and backward pass for them.

- **Flags**: `requires_grad` and `volatile` record state of tensors and ensure that gradients are only computed when needed.
  - `volatile`: If any input is volatile then output is volatile (deprecated)
  - `requires_grad`: If any input requires a gradient then output requires a gradient
Memory Management

• Aggressively frees intermediate variables
• GPUs are a low memory regime, so memory is very valuable (though this is improving)
• Functions maintain pointers to consumers of its output, deleted when nothing needs its output anymore
• Eliminates reference cycles by saving copies of variables without the function pointer
Critique

• Dangers of in place operations not well explored in the paper
• Performance results for Pytorch as opposed to other frameworks