CSS553: Software Architecture
G2: Recovering Architecture Styles/Patterns

Open Source Software: PyTorch

Team: Fahmeedha Azmathullah, Fiona Victoria, Sreja B, and Srilekha Bandaru

April 28, 2022

University of Washington, Bothell

Table of Contents

I. INTRODUCTION 2

A. Open Source Project 2

B. Goals 2

C. Sponsoring organization 2

D. Other Information 2

II. ARCHITECTURE

A. Principal Design Decisions (PDD) 3

B. High-level architectural styles/patterns 3

C. Module: nn 5

1. Architectural styles/patterns 6

2. Specific code component 6

D. Module: ao 7

1. Specific code component 7

2. Architectural styles/patterns 8

E. Module: autograd 9

1. Architectural styles/patterns 10

2. Specific code component 10

F. Module: jit 11

1. Architectural styles/patterns 12

2. Specific code component 13

G. Alignment with Principal Design Decisions 14

III. REFLECTION

A. Steps involved in recovering architectural styles/patterns 16

B. Improving the design recovery process 16

C. Time spent 16

IV. REFERENCES 16

I. Introduction

A. Open Source Project

Open-source project - PyTorch

PyTorch is an open-source machine learning framework developed by Facebook's AI Research lab (FAIR) that uses a python interface for applications such as machine learning, deep learning, natural language processing, and computer vision [1]. It provides python packages for high-level features like tensor computation with strong GPU acceleration and Deep neural networks built on a tape-based automatic differentiation system.

B. Goals

PyTorch, a python-based scientific computing package serves two broad purposes:

1. A replacement for NumPy to use the power of GPUs and other accelerators.
2. An automatic differentiation library that is useful to implement neural networks.

C. Sponsoring organizations

Sponsoring organizations - Facebook, Google, AWS, and Microsoft [1]

D. Other information

Facebook operates both *PyTorch* and *Convolutional Architecture for Fast Feature Embedding* (Caffe2), but models defined by the two frameworks were mutually incompatible. The Open Neural Network Exchange (ONNX) project was created by Facebook and Microsoft in September 2017 for converting models between frameworks. Caffe2 was merged into PyTorch at the end of March 2018.
Around the world in 2022, over 5699 companies have started using PyTorch as a Data Science And Machine Learning tool. Some of the customers include Pluralsights, Grid Dynamics, Digital Ocean, Cambricon Technologies, and Scaleway.

Pytorch Origin

PyTorch began as an internship project by Adam Paszke in October 2016. PyTorch then got two more core developers on board and around 100 alpha testers from different companies and universities. However, the core original authors were Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan.

Users

- AllenNLP is an open-source research library built on PyTorch for designing and evaluating deep learning for NLP.
- ELF is a platform for game research that allows developers to test and train their algorithms in various game environments.
- Stanford University uses Pytorch in its research of new algorithmic approaches.
- Udacity uses PyTorch to educate the next wave of AI innovators.
- Salesforce uses PyTorch to push the state of the art in NLP and Multi-task learning.

Releases

PyTorch has released 21 versions since 2016. The rate at which releases were being made was based on the stage of development [1].

- At the initial stage, Pytorch was busy developing new functions, so it released a new version every month, or even twice a month.
- In the second stage, Pytorch tended to be stable, so it was released nearly every two months.
- In the recent stage, Pytorch focused more on fixing existing bugs than developing new functions, so it was released at a frequency lower than before.

II. Architecture

A. Principal Design Decisions

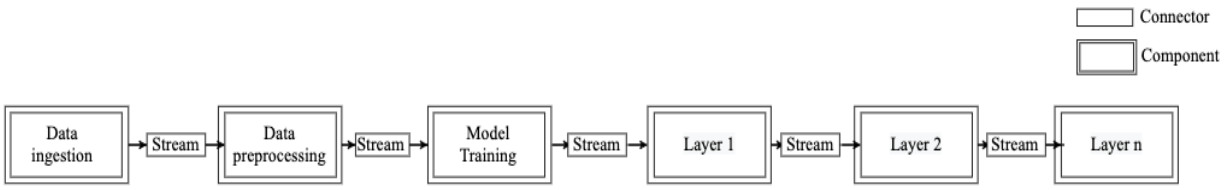
PyTorch's success stems from weaving previous ideas into a design [3] that balances speed and ease of use.

1. Usability-centric design - Deep learning models are just Python programs
2. Interoperability
3. Extensibility
4. Automatic differentiation
5. An efficient C++ core
6. Separate control and data flow
7. Custom caching tensor allocator
8. Multiprocessing
9. Reference counting
10. Be Pythonic
11. Put researchers first
12. Provide pragmatic performance
13. Worse is better

B. Architectural styles/patterns

Some of the identified architectural styles/patterns are as follows,

1. Pipe and filter



ML pipelines are a means of automating the machine learning workflow by enabling data to be collected, transformed, and correlated into a model that can then be analyzed to achieve outputs. This

step-by-step process is typically designed in a pipe and filter architecture style which allows for user interactivity, feedback loops, is easy to reuse, enables modularity, and parallelism.

Fig 1. PyTorch implementation in Pipe and Filter style

2. Object-oriented

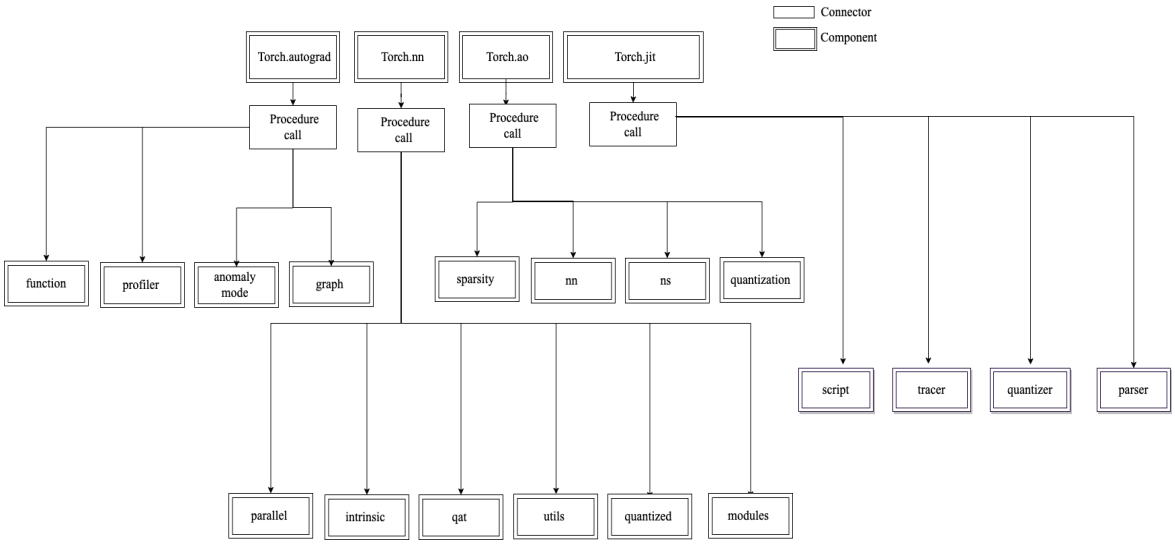


Fig 2. PyTorch - Object-oriented architectural style

The usage of classes in Python for different modules of PyTorch is the primary reason why the architectural style implemented is object-oriented. PyTorch follows an Object-Oriented Programming design and hence, there are classes using which objects can be created to build-train-deploy the machine learning models.

3. Layered

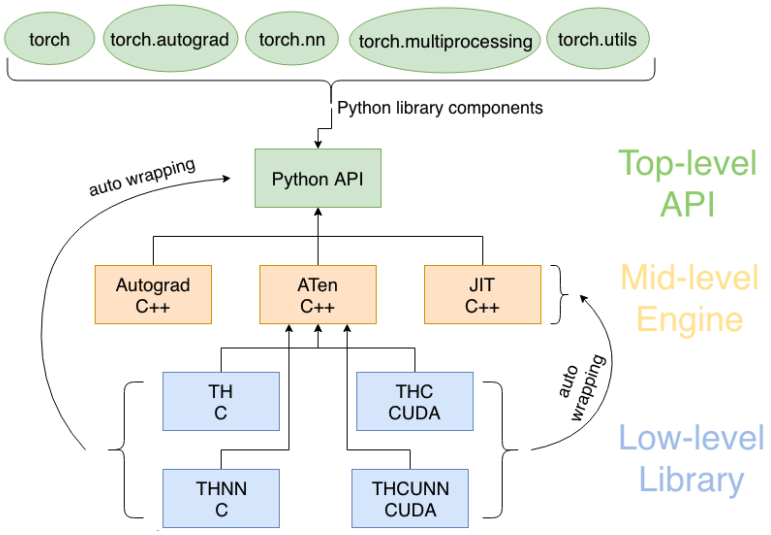


Fig 3. PyTorch - Layered architectural style [1]

The top-level Python library of PyTorch exposes an easy-to-understand API for users to quickly perform operations on tersors, build and train a deep neural network. The engines on the middle-level of module structure consist of autograd, JIT compiler and ATen that are written in C++. The low-level C or CUDA library does all the intensive computations assigned by the upper-level API.

The low-level libraries can be utilized not only by its standard wrapper ATen but also top-level Python APIs and mid-level engines by means of auto wrapping. This kind of architecture keeps the code loosely coupled, decreasing the overall complexity of the system and encouraging further development.

C. Module: nn

Torch.nn module, the heart of PyTorch deep learning is a neural networks library deeply integrated with autograd designed for maximum flexibility. Pytorch uses a torch.nn base class which can be used to wrap parameters, functions, and layers in the torch.nn modules [4]. The module provides basic building blocks for constructing graphs that are used for deep learning training, evaluation, and inference phases. When any deep learning model is developed using the subclass of the torch.nn module, it uses a method like forward(input) which returns the output. A simple neural network takes the input to add weights and bias to it feed the input through multiple hidden layers and finally returns the output.
Some of the major modules and classes provided by torch.nn are mentioned below,

Parameters

The torch.nn module provides a class torch.nn.Parameter() as a subclass of Tensors. If tensors are used with Module as a model attribute, then it will be added to the list of parameters. This parameter class can be used to store a hidden state or learnable initial state of the RNN model.

Containers

Container classes are used to create complex neural networks. Containers use nn.Container() class to develop models. It is a base class to create all neural network modules. Modules are serializable and may have other modules added to the model which forms a tree-like structure.

Layers

Pytorch provides different modules in torch.nn to develop neural network layers. Different trainable layers can be configured using a respective class from torch.nn Module. Some of the commonly used layers include Convolution, Pooling, Padding, Normalization, Transformer etc.

Functions

Module supports the classes for the various distance and loss functions

Apart from the modules and classes mentioned above, there are several other classes and modules to perform data parallelism, distributed data parallelism, non-Linear Activations, and utilities.

Architectural style for module nn - *Object-oriented*

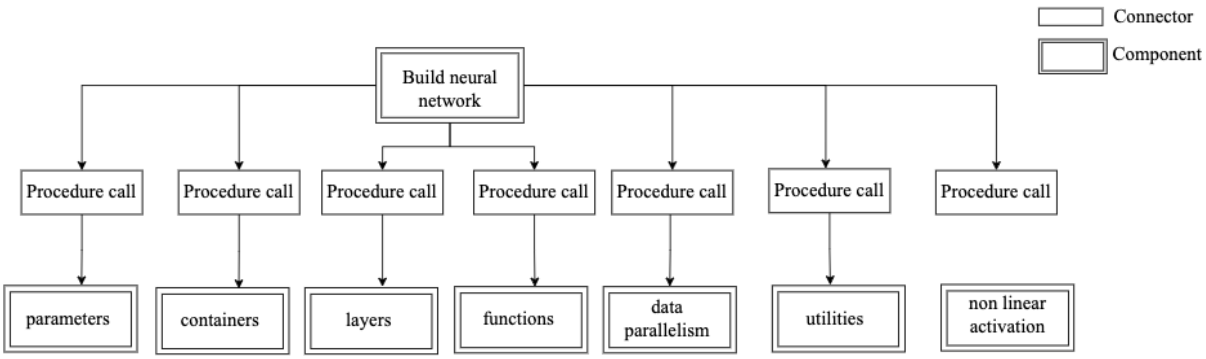


Fig 4. Architectural style for module: nn

Specific code that corresponds with the components

- Build neural networks - https://github.com/pytorch/pytorch/tree/master/torch/nn
- Parameters - https://github.com/pytorch/pytorch/blob/master/torch/nn/parameter.py#L6
- Containers - https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/container.py#L16
- Layers - Convolution, MaxPool, and Padding
- Functions - L1 Loss function, Cross Entropy loss function
- Data parallelism - Data parallel and distributed data parallelism
- Utilities - Prune entire channels in a tensor at random etc
- Non-linear activations - RELU, Tanh and LeakyRelu

D. Module: ao

Quantization:

Quantization refers to techniques for doing both computations and memory accesses with lower precision data, usually int8 compared to floating-point implementations [5]. Fundamentally quantization means introducing approximations and the resulting networks have slightly less accuracy. These techniques attempt to minimize the gap between the full floating-point accuracy and the quantized accuracy. There are three techniques for quantizing neural networks in PyTorch. They are Dynamic Quantization, Post-Training Static Quantization, and Quantization Aware Training.

Dynamic Quantization:

This involves converting the weights to int8, and converting the activations to int8 on the fly, just before doing the computation ("dynamic"). The computations will thus be performed using efficient int8 matrix multiplication and convolution implementations, resulting in faster computations

Post-Training Static Quantization:

The performance (latency) can be further improved by converting networks to use both integer arithmetic and int8 memory accesses. Static quantization performs the additional step of first feeding batches of data through the network and computing the resulting distributions of the different activations. This additional step allows us to pass quantized values between operations instead of converting these values to floats - and then back to ints - between every operation, resulting in a significant speed-up.

Quantization Aware Training (QAT):

The QAT method typically results in the highest accuracy of these three. With QAT, all weights and activations are "fake quantized" during both the forward and backward passes of training: that is, float values are rounded to mimic int8 values, but all computations are still done with floating-point numbers. Thus, all the weight adjustments during training are made while "aware" of the fact that the model will ultimately be quantized; after quantizing, therefore, this method usually yields higher accuracy than the other two methods.

Specific code that corresponds with the components:

Dynamic quantization -

- https://github.com/pytorch/pytorch/blob/torch/ao/quantization/quantize.py

Post-training static quantization -

- https://github.com/pytorch/pytorch/blob/torch/ao/quantization/fuse_modules.py#L84
- https://github.com/pytorch/pytorch/blob/torch/ao/quantization/quantize.py#L231
- https://github.com/pytorch/pytorch/blob/torch/ao/quantization/quantize.py#L487

Quantization Aware Training (QAT) -

- https://github.com/pytorch/pytorch/blob/torch/ao/quantization/quantize.py#L437
- https://github.com/pytorch/pytorch/blob/torch/ao/quantization/quantize.py#L487

Architectural styles/patterns for quantization: *Layered - Client Server.*
Note: Connectors are procedure calls

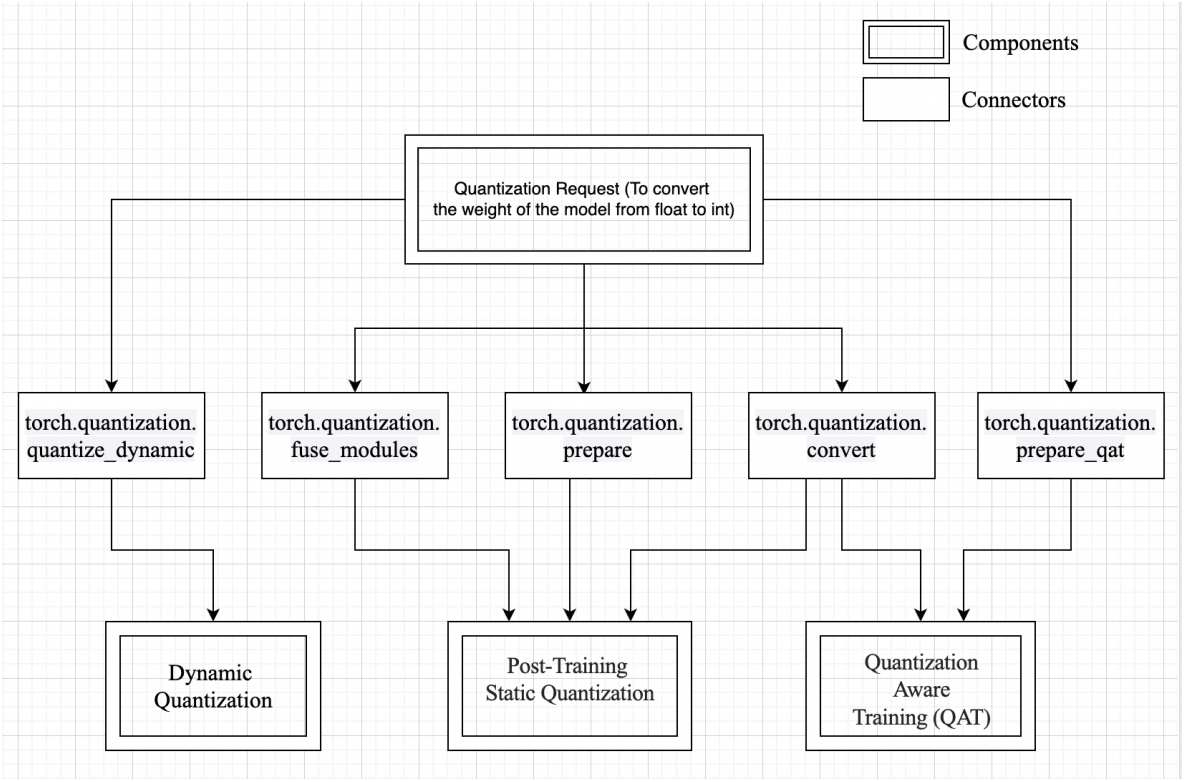


Fig 5. Architectural style for quantization in module: ao

E. Module: autograd

Autograd is a reverse **automatic differentiation system**. It is a hotspot for PyTorch performance, so most of the heavy lifting is implemented in C++. This implies that we have to do some shuffling between Python and C++; and in general, data is in a form that is convenient to manipulate from C++. The general model is that for any key data type that autograd manipulates, there are two implementations: a C++ type and a Python object type.

Torch layer accesses services provided by autograd layer which in turn uses the interfaces provided by autograd-C++ layer. The csrc directory contains all of the code concerned with integration with Python. This is in contrast to lib, which contains the Torch libraries that are Python agnostic. csrc depends on lib, but not vice versa. Python.h has the python variable definition.

Components:

1. **Anomaly detection:** Context-manager that enables anomaly detection for the autograd engine.
2. **Forward-mode automatic differentiation:** It can be used to compute a directional derivative by performing the forward pass as before, except it associates the input with another tensor representing the direction of the directional derivative (or equivalently, the v in a Jacobian-vector product).
3. **Gradient Computation:** Context managers for enable, disable and set (on/off) gradient calculation.
4. **Inference mode:** Enables or disables inference mode.
5. **Numerical gradient checking:**
    - Check gradients computed via small finite differences against analytical gradients.
    - Check gradients of gradients computed via small finite differences against analytical gradients.
6. **Profiler:** Context manager that manages autograd profiler state and holds a summary of results. Under the hood it just records events of functions being executed in C++ and exposes those events to Python. You can wrap any code into it and it will only report runtime of PyTorch functions. There's no way to force nvprof to flush the data it collected to disk, so for CUDA profiling one has to use this context manager to annotate nvprof traces and wait for the process to exit before inspecting them.
7. **Saved Tensors:**
    - **Saved_tensors_hooks:** Context-manager that sets a pair of pack / unpack hooks for saved tensors.
    - **Save_on_cpu:** Context-manager under which tensors saved by the forward pass will be stored on cpu, then retrieved for backward.

Architectural Style/ Pattern: Layered Architecture - Virtual Machine

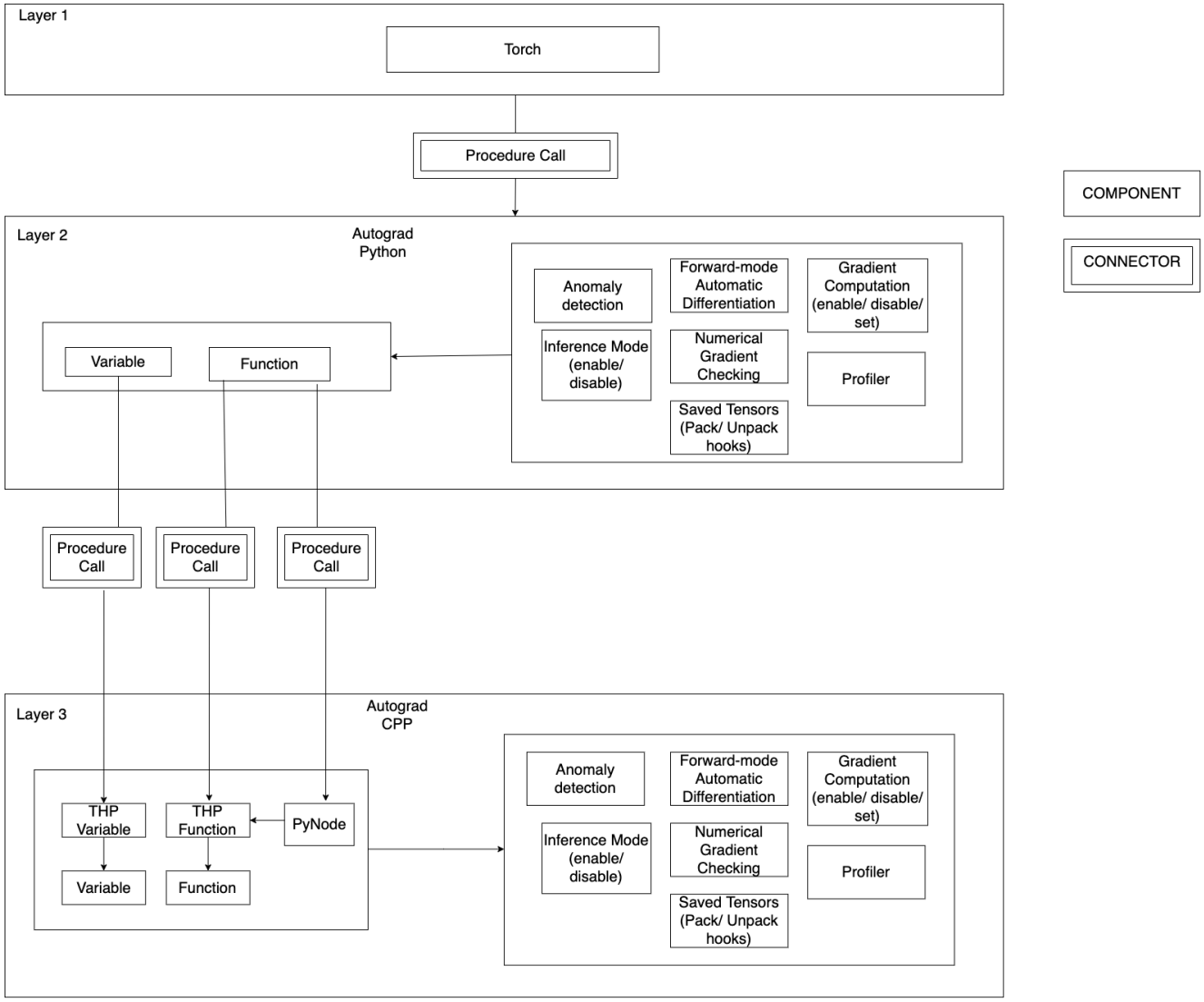


Fig 6. Architectural style for module: autograd

Specific code that corresponds with the components:

- **Autograd Python**
    1. Variable: Variable class
    2. Function: Function class
    3. Anomaly Detection: Detect Anomaly, Set anomaly detection (on/off)
    4. Forward-mode Automatic Differentiation: dual_level
    5. Gradient Computation: Enable gradient calculation, Disable gradient calculation, Set gradient calculation (on/off)

6. Inference Mode: Enables or disables inference mode
7. Numerical gradient checking: Checks gradients and gradients of gradients
8. Saved Tensors: pack/unpack hooks
9. Profiler: autograd profiler state

- **Autograd CPP**

10. THP Variable: Torch Python variable
11. THP Function: Torch Python Function
12. PyNode: Subclass of node
13. Variable: Variable is the same as a tensor
14. Function: All functions in PyTorch's autograd machinery derive from this class
15. Anomaly Detection: Anomaly Detection
16. Forward-mode Automatic Differentiation: forward_grad
17. Gradient Computation: grad_mode
18. Saved Tensors: Saved variable hooks
19. Profiler: profiler.h

F. Module: jit (Just-In-Time):

Jit is an *optimized compiler* that is used to compile PyTorch modules/programs. Two of the main design principles of jit compiler are to **enable static typing** and **structured control flow**. It helps in improving efficiency and performance by building PyTorch models. Here models are object-oriented python programs. PyTorch basically supports 2 modes to handle research and production. Pytorch can run in 2 modes: *Eager mode & Script mode.* [7]

Eager mode: normal python runtime mode for port which is used for faster prototyping, training, and experimentation.
Script mode: it basically has two major components which are PyTorchJIT and TorchScript.
TorchScript runtime is our own runtime (different from python runtime) allows us to run things threaded in *parallel* and lets us do *performance optimizations*.

Trace module converts eager mode to script mode but only preserves tensor operations during the conversion.
Script module uses torch.jit.script() to directly parse the python code.

Two of the major components in pytorch jit are torchscript and jit compiler, below are the subcomponents that fall under the major components.

Components: script, tracer , quantizer, builder
Connectors: quantizer(torch.ao.quantization.quantize_dynamic)
script(torch.jit.script)
Tracer (torch.jit.trace)
parser- (torch.jit.trace)

Architectural style(s)/pattern(s): *Layered(Client - server) and Main Program-Subroutine*
Note: double boxed - components; single box with gradient- connectors

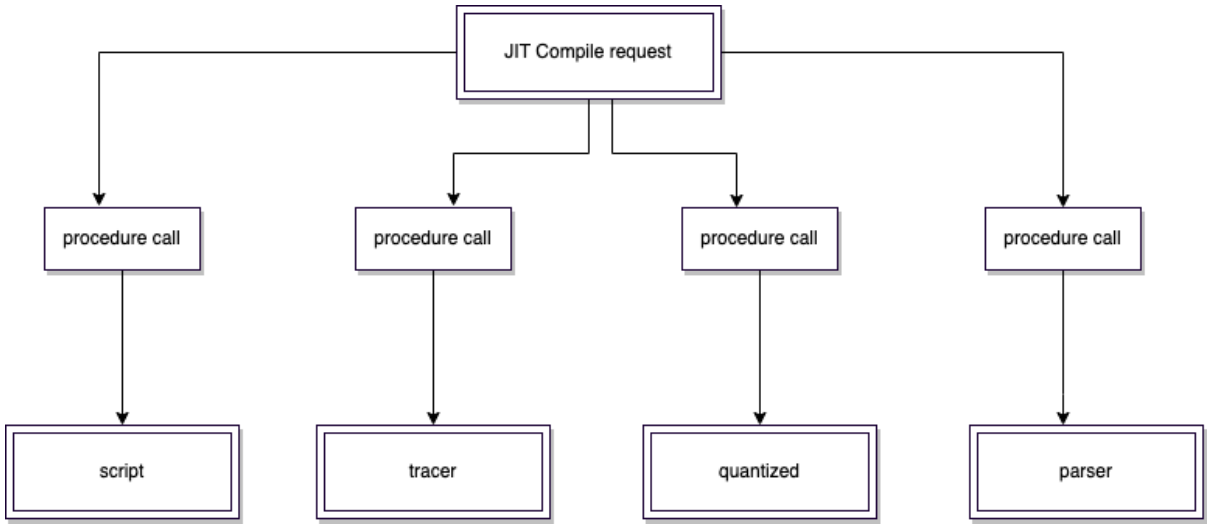


Fig 7. Architectural style for module: jit

Tracer: It acts like a tool which takes the eager model and user inputs as inputs, run the eager model and will record the tensor operations that take place during the time at which the model is running. The recording is then converted into torchscript module. On a high level tracer produces graphs by recording what operations
are performed on tensor by using torch.jit.trace api.

Script: It will directly parse the python code and will convert the code into modules. The conversion happens in two phases, first phase is where an abstract syntax tree(AST) is created and this tree is created by making use of tree objects. Then the IR emitter does semantic analysis and abstracts the tree into a module. The conversion takes place when torch.jit.script api is called.

Parser: It uses lexer to build the abstract syntax tree, in jit parsing is done in a recursive descent manner. It resolves operator precedence issues of an expression by augmenting the function with a precedence variable p , which means when the function is called it parses an expression whose operators all have precedence higher than p.

Quantizer: After generating the torchscript modules, they should be optimized, which is where the quantizer component is used. It follows a technique where it performs the operations and stores the tensors at lower bandwidths. This quantization is done on multiple neural networks like GRU Cells, LSTM, RNN CellBase etc.
To perform quantization in jit we use torch.ao.quantization.quantize_dynamic api.

Overall there are multiple components in the Pytorch jit module, but we felt that the above components played a major role in the jit module.

Specific code that corresponds with the components:

- **Tracer:** **code for torch.jit.tracer**
- **Script**: **code for torch.jit.script**
- **Pasrer:** **code for parser**
- **Quantizer:** **code for quantizer**

G. Alignment with the principal design decision

1. ***Usability-centric design*:**

Making use of object-oriented style and separating the PyTorch modules into three levels (upper-middle-lower) enables easier interaction between users and the library.
Another example can be referred from the torch.nn module wherein the layers, composing models, loading data, running optimizers, and parallelizing the training process are all expressed using the familiar concepts developed for general-purpose programming (python classes). By doing so, any new potential neural network architecture can be easily implemented with PyTorch.

2. ***Interoperability*:**

Pytorch supports *__cuda_array_interface__,* so zero-copy data exchange between CuPy and PyTorch can be achieved at no cost. PyTorch allows for bidirectional exchange of data with external libraries. For example, it provides a mechanism to convert between NumPy arrays and PyTorch tensors using the torch.from_numpy() function and .numpy() tensor method.

3. ***Extensibility*:**

The automatic differentiation system allows users to add support for custom differentiable functions.
To do that users can define a new subclass of torch.autograd.Function that implements forward()
and backward() methods, which specify the function and its derivative.

4. ***Automatic differentiation:***

This is a library designed to enable rapid research on machine learning models. It provides a high performance environment with easy access to automatic differentiation of models executed on different devices (CPU and GPU). Although PyTorch's support for automatic differentiation was heavily inspired by its predecessors (especially twitter-autograd and Chainer), it introduces some novel design and implementation choices, which make it the one of the fastest implementations among automatic differentiation libraries. [8]

- **In-place operations:**

  An In-place operations pose a hazard for automatic differentiation, because

1) An in-place operation can invalidate data that would be needed in the differentiation phase.
2) They require non trivial tape transformations to be performed.

   PyTorch implements simple but effective mechanisms that address both of these problems. [8]

- **No tape:**

PyTorch users can mix and match independent graphs however they like, in whatever threads they like (without explicit synchronization). An added benefit of structuring graphs this way is that when a portion of the graph becomes dead, it is automatically freed; an important consideration when we want to free large memory chunks as quickly as possible. [8]

- **Core logic in c++:**

Carefully tuned C++ code is one of the primary reasons PyTorch can achieve much lower overhead compared to other frameworks. [8]

5. ***An efficient C++ core*:**

Placing a helper function (a function that does not need direct access to the representation of the class, yet is seen as part of the useful interface to the class.) in the same namespace is one of the C++ core guidelines. *_propagate_qconfig_helper* function in the quantization component aligned with this PDD.

6. ***Separate control and data flow*:**

Python has a strict separation between data flow (tensors and operations performed on them) and control flow(loops and data processing). Control flow is handled by python and optimized C++ code is hosted on the CPU, and results in the linear sequence of operator invocations are parsed by jit parser. These operators are executed asynchronously on GPU by leveraging the CUDA Stream mechanism. This causes the system to overlap the python code on CPU with tensor operators on GPU, as tensor operations take a significant amount of time, and GPU reaches its peak performance. This mechanism is invisible to users unless they use multi-stream primitives.

7. ***Custom caching tensor allocator*:**

PyTorch implements a custom allocator which incrementally builds up a cache of CUDA memory and reassigns it to later allocations without further use of CUDA APIs. The purpose of this implementation is to avoid the bottleneck caused where the cudaFree routine may block its caller until all previously queued work on all GPUs completes. This is present in the low-level CUDA library of the PyTorch framework.

8. ***Multiprocessing*:**

Pytorch provides the torch.multiprocessing package which is a replacement for Python's multiprocessing module. It supports the exact same operations, but extends it, so that all tensors sent through a multiprocessing.Queue, will have their data moved into shared memory and will only send a handle to another process. This allows the implementation of various training methods, like Hogwild, A3C, or any others that require asynchronous operation.

9. ***Reference counting*:**

In order to deliver great performance, the PyTorch library treats memory as a scarce resource that it needs to manage carefully. Reference counting is a common memory management technique in C++ but PyTorch does its reference counting in a slightly different way using intrusive_ptr. intrusive_ptr is an alternative to shared_ptr that has better performance because it does the refcounting intrusively
(i.e. in a member of the object itself).

10. *Building for scale*:

PyTorch essentially optimized the common patterns in neural networks and improved inference latency and throughput.

III. Reflection

A. Steps for recovering Architecture styles/patterns

- Read through Pytorch documentation and code trace of Github to get the main components.
- Read through the research papers to get an overall understanding of Pytorch and its architecture.
- Used the Visual paradigm diagram (Reverse Engineering tool) to find more of the sub-components within the main components.
- Examined the code in GitHub to find out how each component is connected/called.
- Looked for architectural styles and patterns within each component.

B. Improving the design recovery process

- Understanding the specific domain (ML) to fasten the process of design recovery
- Analyze the data and control flow more and see its difference in implementation

C. Total time spent: ~ 20 - 25 hours

Fahmeedha - 5 hours
Srilekha - 5 hours
Sreja - 6 hours
Fiona - 6 hours

IV. References

1. *PyTorch*. Code of Honour. Retrieved April 11, 2022, from TU Delft https://se.ewi.tudelft.nl/desosa2019/chapters/pytorch/
2. *PyTorch* (2016). https://github.com/pytorch/pytorch
3. Adam, P., et al (Dec 2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library https://arxiv.org/pdf/1912.01703.pdf
4. *Introduction to torch.nn Module.* Retrieved April 26, 2022, from https://www.educba.com/torch-dot-nn-module/
5. Quantization in Pytorch https://pytorch.org/docs/stable/quantization.html#quantization-doc
6. *PyTorch Documentation.* Retrieved April 11, 2022, from https://pytorch.org/docs/stable/
7. *PyTorch JIT and TorchScript.* Retrieved April 14, 2022, from https://towardsdatascience.com/pytorch-jit-and-torchscript-c2a77bac0fff
8. *Automatic Differentiation in PyTorch* https://openreview.net/pdf?id=BJJsrmfCZ