G1: As-implemented vs As-intended Architectures of PyTorch

Team: Fahmeedha Azmath, FionaVictoria, Sreja B and Srilekha Bandaru
Open source link: https://github.com/pytorch/pytorch
UML Generation Tool: Visual Paradigm - Python

PyTorch is an open source machine learning framework developed by Facebook's AI Research lab (FAIR) that uses python interface for applications such as machine learning, deep learning, natural language processing and computer vision. It provides a python package for high level features like tensor computation with strong GPU acceleration.

As-Intended Architecture

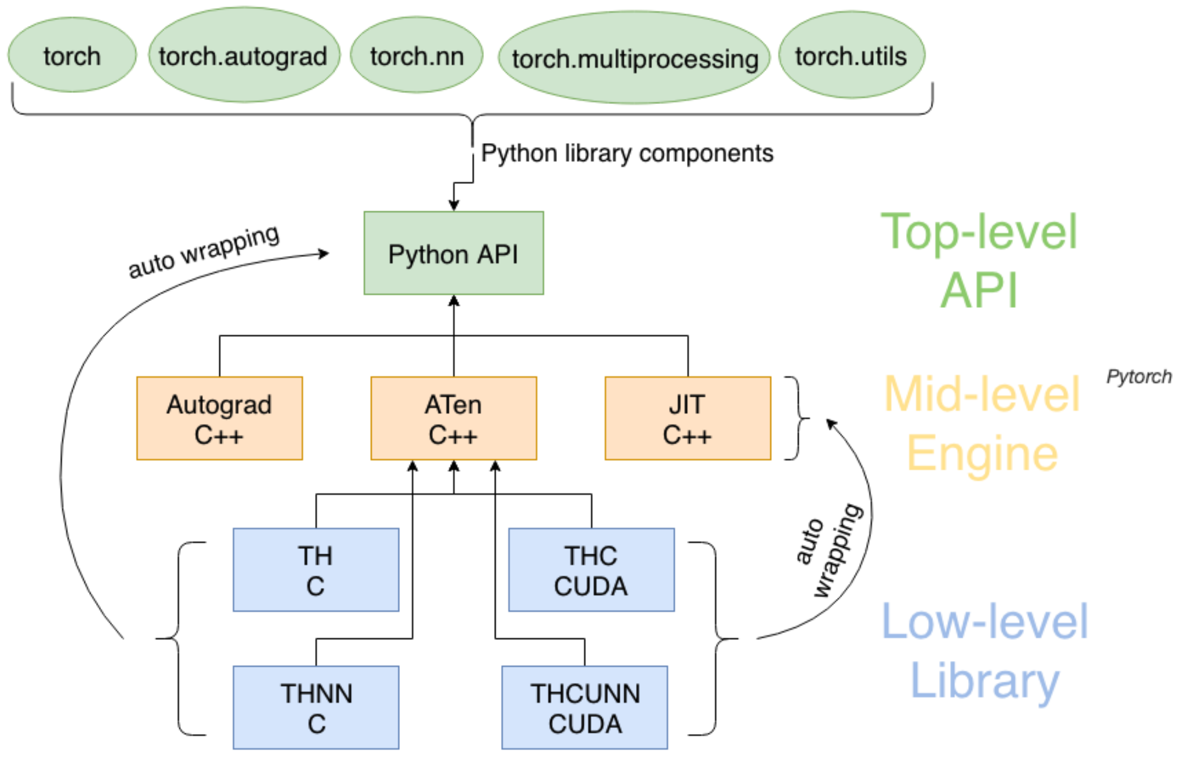

Fig 1. High level architecture - PyTorch [1]

The foundation of Pytorch relies on a core data structure, the Tensor, whose implementation is similar to that of Numpy arrays. From that foundation, a list of features have been built making it easy to get a project up and running, or an investigation into a new neural network architecture designed and trained. Tensors provide acceleration of mathematical operations. PyTorch has packages for distributed training, worker processes for efficient data loading, and an extensive library of common deep learning functions.

The core PyTorch modules for building neural networks provide common neural network layers and other architectural components like fully connected layers, convolutional layers, activation functions, and loss functions.

As-Implemented Architecture

Fig 2. UML diagram of As-Implemented Architecture - PyTorch

Full architecture image link:
https://drive.google.com/file/d/1z2FWrmHjdy0XVWnnHYiB9WjeFq59HHVX/view?usp=sharing

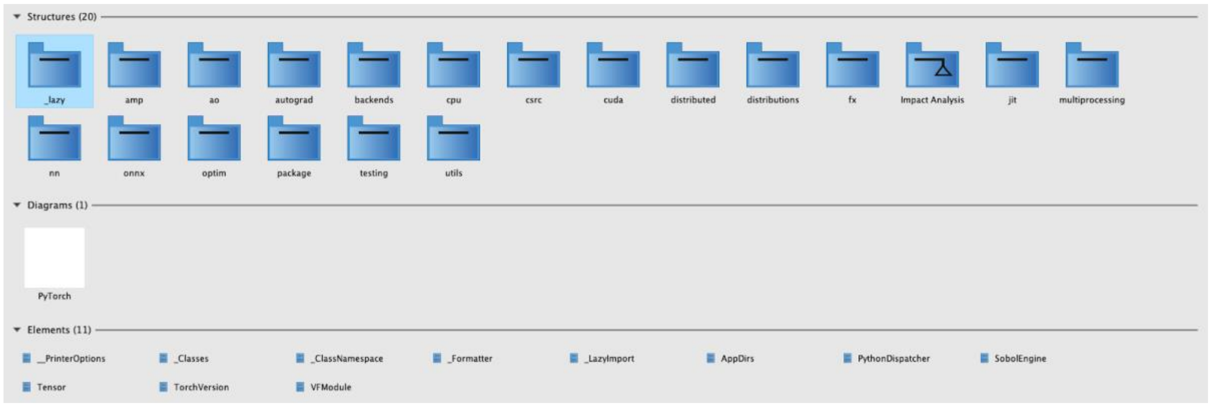

Fig 3. Model Structure

The similarities and differences between the major packages/modules of Pytorch are discussed below:

1. Multiprocessing

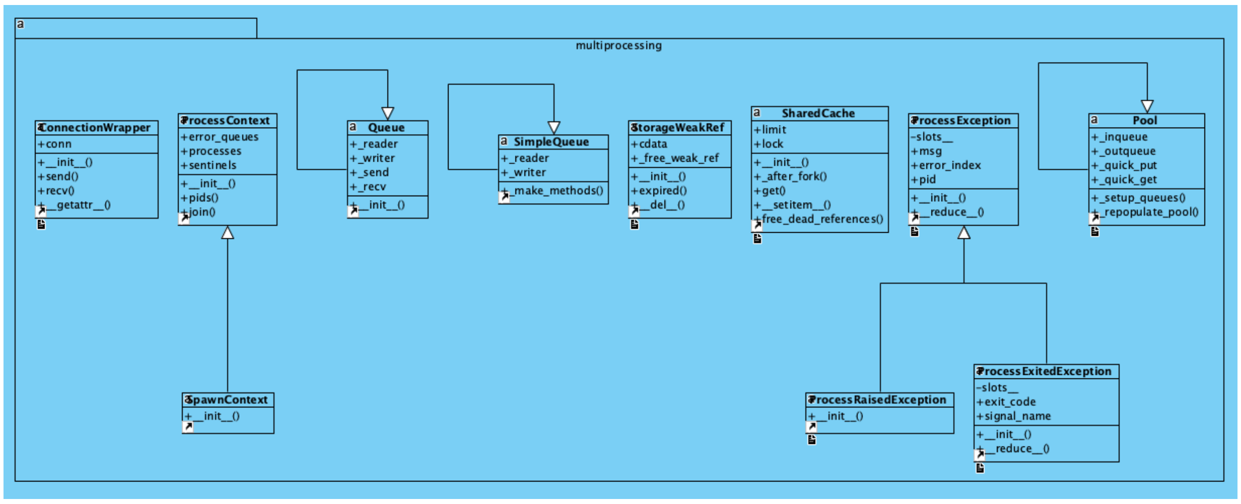

Fig 4. Multiprocessing module UML Layout

Multiprocessing is a package that supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using sub processes instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows. [2]

torch.multiprocessing is a wrapper around the native multiprocessing module. It registers custom reducers that use shared memory to provide shared views on the same data in different processes. Once the tensor/storage is moved to shared_memory, it will be possible to send it to other processes without making any copies [3].

Similarities

- **get_context()** is used in both architectures to get a context object. Context objects have the same API as the multiprocessing module. It allows multiple start methods in the same program.
- Both **Queue** and **SimpleQueue** classes use the methods of queue. Queue. Only difference is that the SimpleQueue class only reads and writes whereas the Queue class reads, writes, sends and receives.

Differences

- Original documentation provides three ways to start the process which includes **spawn, fork, and forkserver,** however the current implemented version of pytorch includes only spawn start process. This is because spawn supports the sharing of CUDA tensors between processes (CUDA compatible).
- There are two types of communication channels between processing which are supported by multiprocessing in original architecture. This includes **Queues and Pipes**. However, current architecture uses only a queue mechanism to move tensors to shared memory.
- Other than queue, there are also SimpleQueue and JoinableQueue which are FIFO queues on the queue class in the standard library. Current architecture does not use **JoinableQueue** since it needs to call done() for every completed task. Otherwise overflow exceptions will arise.

2. Autograd

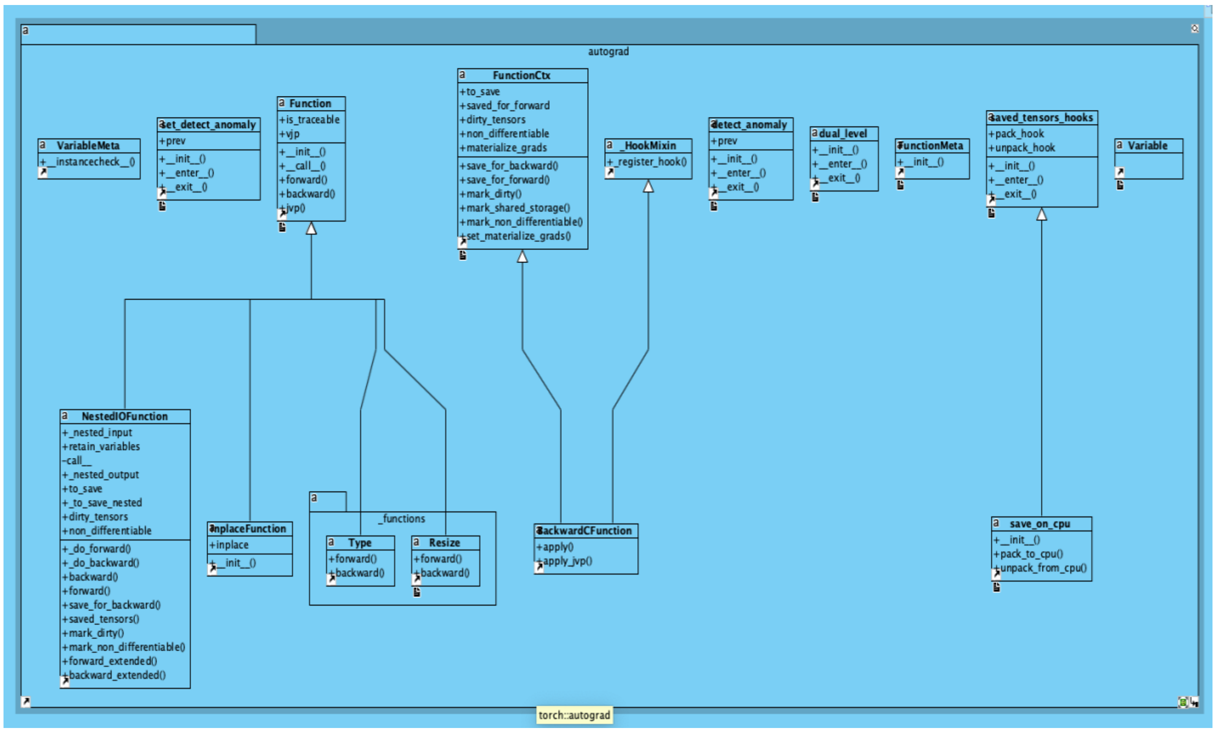

Fig 5. Autograd package UML Layout

PyTorch’s *Autograd* feature is part of what makes PyTorch flexible and fast for building machine learning projects. It allows for the rapid and easy computation of multiple partial derivatives (also referred to as *gradients)* over a complex computation. This operation is central to backpropagation-based neural network learning. The power of autograd comes from the fact that it traces the computation dynamically *at runtime,* meaning that if the model has decision branches, or loops whose lengths are not known until runtime, the computation will still be traced correctly, and correct gradients to drive learning are obtained. This, combined with the fact that models are built in Python, offers far more flexibility than frameworks that rely on static analysis of a more rigidly-structured model for computing gradients [6].

torch.autograd is a tape based automatic differentiation library that provides classes and functions to support all differentiable Tensor operations in torch. It requires minimal changes to the existing code - only needed to declare Tensor s for which gradients should be computed with the requires_grad=True keyword. As of now, it only supports autograd for floating point Tensor types (half, float, double and bfloat16) and complex Tensor types (cfloat, cdouble). It also powers neural network training [7].

Similarities:

1. Function: It provides Function and FunctionCtx classes. Function is the Base class to create custom autograd.Function.
2. Profiler:
    - Profile class (CPU only): Context manager that manages autograd profiler state and holds a summary of results. It records events of functions being executed in C++ and exposes those events to Python. It provides export_chrome_trace( ), key_averages( ), self_cpu_time_total( ) and total_average( ).
    - Emit_nvtx class (CPU and GPU): It provides load_nvprof( ) that opens an nvprof trace file and parses autograd annotations.
3. anomaly_mode: It provides detect_anomaly class that enables anomaly detection for autograd engines and set_detect_anomaly class that sets the anomaly detection on/off.
4. graph:
    - Saved_tensors_hooks: This context manager is used to define how intermediary results of an operation should be packed before saving, and unpacked on retrieval.
    - Save_on_cpu: This context manager is used to trade compute for GPU memory usage.
5. dual_level: Context-manager that enables forward AD. All forward AD computation must be done in a dual_level context. Methods that are similar in both the as-indented and as-implemented architectures are make_dual( ) which associates a tensor value with a forward gradient to create a “dual tensor” to compute forward AD gradients and unpack_dual( ) that unpacks a “dual tensor” to get both its Tensor value and its forward AD gradient.
6. functional: It contains the higher level API for the autograd that builds on the basic API above and allows the computation of jacobian, hessian, vjp, jvp, vhp, hpv in both the as-indented and as-implemented architectures. This API works with user-provided functions that take only Tensors as input and return only Tensors.

7. grad_mode: It contains no_grad class, enable_grad class, set_grad_enabled class and inference_mode class
8. gradcheck: It provides gradcheck( ) and gradgradcheck( ) to check gradients and gradients of gradients respectively.

Differences:

1. dual_level: enter_dual_level( ), exit_dual_level( ) methods are available in as-implemented architecture, but not in intended architecture.
2. grad_mode: It contains _DecoratorContextManager class that is present only in as-implemented architecture.
3. Function: _HookMixin, BackwardCFunction, FunctionMeta, InplaceFunction, NestedIOFunction classes are only available in as-implemented architecture. The FunctionCtx class provides save_for_forward( ) and mark_shared_storage( ) that are only available in as-implemented architecture.
4. gradcheck: It provides get_numerical_jacobian to compute numerical jacobians and get_numerical_jacobian_wrt_specific_input that computes numerical jacobians to a single input and returns N jacobians, where N is the number of outputs.
5. Profiler: It provides record_function class which is a context manager decorator that adds a label to a block of Python code when running autograd profiler. It is useful when tracing the code profile. It also provides an EnforceUnique class which raises an error if a key is seen more than once. These classes are available only in as-implemented architecture. Profile class provides exportstacks( ) which is only available in as-implemented architecture.

3. Optim

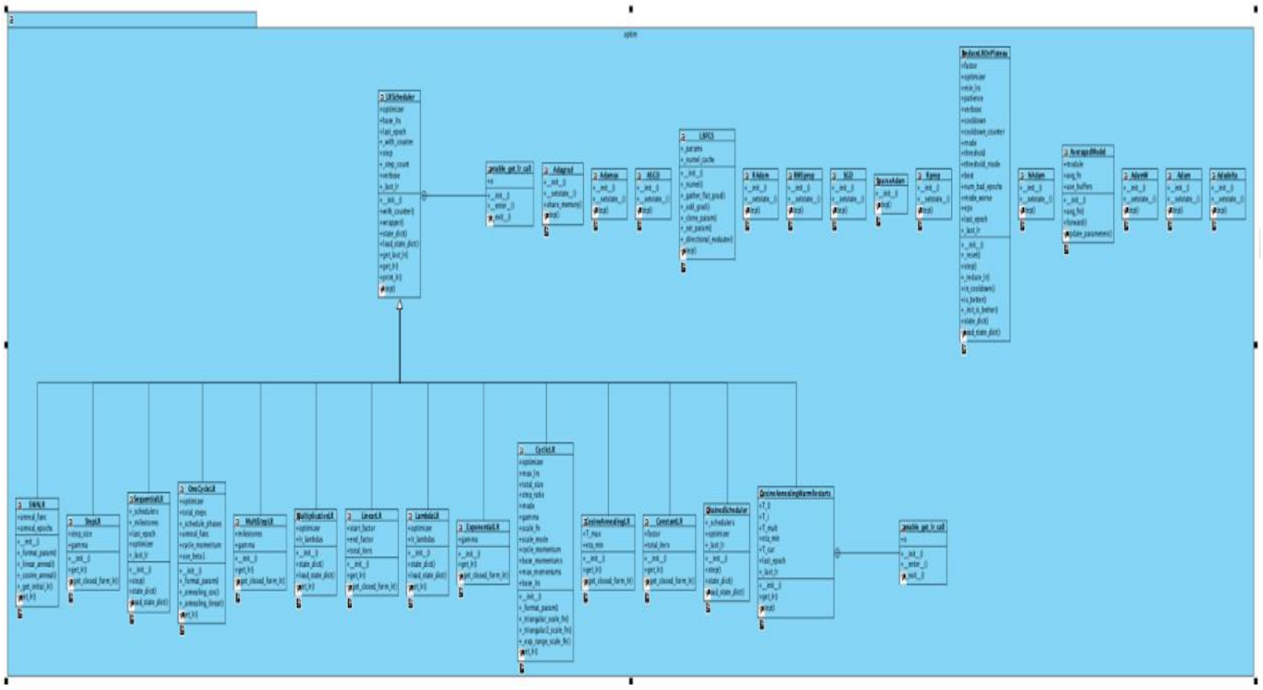

Fig 6. Optim package UML Layout

Optim, a top-level python library in PyTorch provides APIs that users can use to implement optimization algorithms. In order to use this package, users would have

to construct an optimizer object, that will hold the current state and update the parameters based on the computed gradients [3].
When constructing the optimizer object, the iterable parameters (Variables) to optimize is to be provided. This includes optimizers such as learning rate, weight decay etc.

Similarities:

- Both the architectures have the common classes such as:

1. Adadelta
2. AdamW
3. ASGD
4. NAdam
5. Adagrad
6. SparseAdam
7. LBFGS
8. RMSProp
9. Adam
10. Adamax
11. RAdam
12. RProp
13. SGD

The above mentioned classes implement their corresponding optimizing algorithm by making use of parameters like params, lr, beta, rho, weight_decay, momentum_decay etc. These classes are designed to work independently and are isolated from one another.

- Another component named “optim.lr_scheduler” provides several classes to adjust the learning rate based on the number of epochs. Learning rate (lr_scheduler) scheduling should be applied after updating the optimizer algorithm. The inherited lr_schedulers found in both architectures are as follows:

1. LambdaLR
2. ConstantLR
3. ChainedScheduler
4. OneCycleLR
5. MultiStepLR
6. MultiplicativeLR
7. LinearLR
8. SequentialLR
9. CosineAnnealingLR
10. CosineAnnealingWarmRestarts
11. StepLR
12. ExponentialLR
13. ReduceLROnPlateau
14. CyclicLR

Differences:

- The **enable_get_lr_call** class found within the _LRScheduler class is present only in the as-implemented architecture.
- The **single_tensor** and **multi_tensor** functional implementations of each optimizer found in torch::optim are not mentioned in the as-intended architecture; however it is implemented in the latest package of optim. These methods are used for optimizations like Stochastic Gradient Descent, Adaptive Gradient Algorithm etc based on the flag set for **jit.is_scripting()**

4. Jit (Just-in-time):

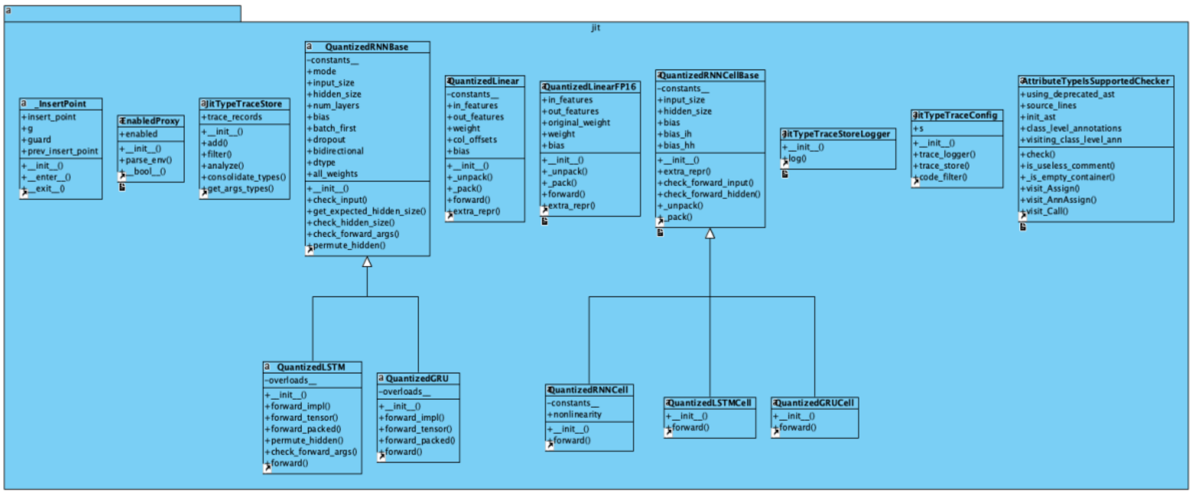

Fig 7. Jit package UML Layout

Jit is basically an *optimized compiler* which is used to compile PyTorch modules/programs. Two of the main design principles of jit compiler are to **improve the developer efficiency** and **building for scale**. It helps in improving the 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.* [8]
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.

Similarities:
Some of the Common classes between the two architectures are:

1. ScriptModule
2. ScriptFunction : Functionally equivalent to ScriptModule, but represents a single function and does not have any attributes or Parameters.

Few of the common modules /functions in both architectures are:

1. *script*: it is a scripting function which will inspect the source code, compile it as TorchScript code using TorchScript compiler and return either of the above 2 classes.
2. *trace*: it will trace the initial root and return the corresponding graph representation which is an executable and will be returned by ScriptFunction.
3. *annotate(the_type, the_value*): it is a pass-through function that returns “the_value” which is used to hint the TorchScript compiler about the type of ”the_value”.

Here the *trace* and *script* modules follow the **script mode,** whereas *annotate* follows **eager mode.**

Differences:
There are many differences between the as-intended arch vs as-implemented arch. As-implemented architecture contains multiple enhancements over the as-intended architecture.

1. Verify : it is a public function which is present in as-implemented architecture, it is a **JIT** compiled model and has the same behavior as its uncompiled version along with its backwards pass. If the model returns multiple outputs, we must also specify a `loss_fn` to produce a loss for which the backwards will be computed
2. run_frozen_optimisation is marked public method in as-Implemented version but doesn't appear anywhere in as-Intended architecture. This method helps in running a series of optimizations looking for patterns that occur in frozen graphs.

Note: Python is not strictly object oriented language, developers are free to implement standalone methods. When using tools which can generate UML diagrams from code, standalone methods won't surface UML because they are not technically part of any class. For example, in jit module UML diagrams didn't contain a lot of as-intended design because as-implemented architecture has normal python functions for script, trace, freeze etc. It is very difficult to figure out differences between both versions of architecture without actually reading through code.

We tried to generate class diagrams using various tools like StarUML and Visual Paradigm but both the tools produced the same results. So, we tried using **pyreverse** which uses **graphviz** to generate package view and class view. Even with pyreverse tool, generated output didn't contain standalone methods.
Here is an example output of class diagram produced for Jit module. (It is very huge and isnt clear) classDiagram_JIT

References

1. *PyTorch*. Code of Honour. Retrieved April 11, 2022, from TU Delft https://se.ewi.tudelft.nl/desosa2019/chapters/pytorch/
2. Native multiprocessing module. Retrieved April 14, 2022, from https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing
3. *PyTorch Documentation.* Retrieved April 11, 2022, from https://pytorch.org/docs/stable/
4. *Deep Learning with PyTorch by Eli Stevens Luca Antiga.* MEAP Publication. Retrieved April 11, 2022, https://livebook.manning.com/book/deep-learning-with-pytorch
5. *How to generate UML from Python.* Updated on June 6, 2019. Retrieved April 11, 2022, from https://circle.visual-paradigm.com/docs/code-engineering/instant-reverse/how-to-generate-uml-from-python/
6. *The Fundamentals of Autograd*. Retrieved April 14, 2022, from https://pytorch.org/tutorials/beginner/introyt/autogradyt_tutorial.html
7. *PyTorch Documentation*: Automatic differentiation package. Retrieved April 14, 2022, from https://pytorch.org/docs/stable/autograd.html
8. *PyTorch JIT and TorchScript.* Retrieved April 14, 2022, from https://towardsdatascience.com/pytorch-jit-and-torchscript-c2a77bac0fff