CSS553: Software Architecture
G3: Design Pattern, Security Pattern, and Implementation Plan

Open Source Software: PyTorch

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

May 19, 2022

University of Washington, Bothell

Table of Contents

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.

Tools used: PyCharm, pyreverse

II. Design Patterns

A. Module: nn - Convolution layer

Design pattern: Bridge

A bridge method is a structural design pattern that allows the separation of large classes into two separate classes (hierarchies). The below-mentioned subsection of module torch.nn uses two different implementations of convolution 2d ie., Conv2d and LazyConv2d. LazyConv2d class is an extension of Conv2d and the two can work independently.

Intent

Decouple an abstraction from its implementation so that the two can vary independently.

Also Known As

Handle/Body

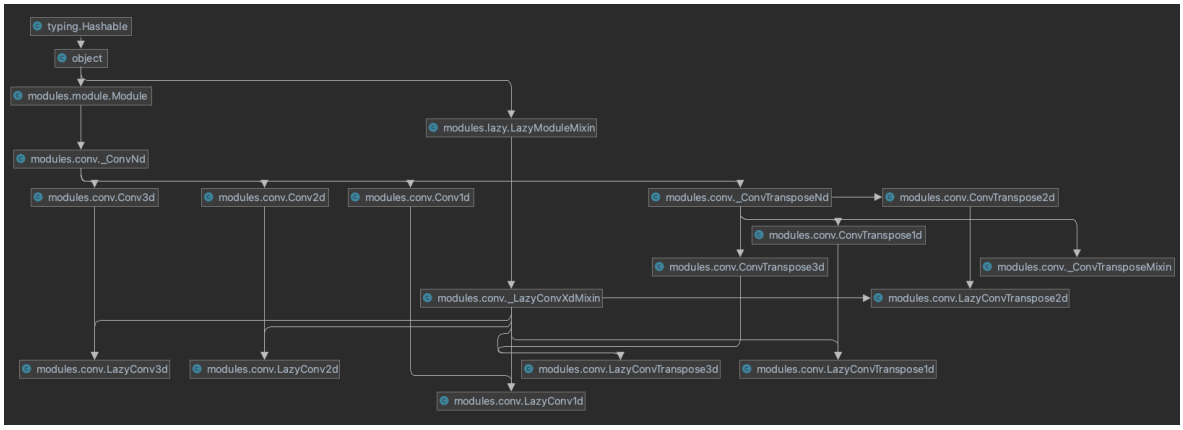

Fig 1. Simplified Class diagram

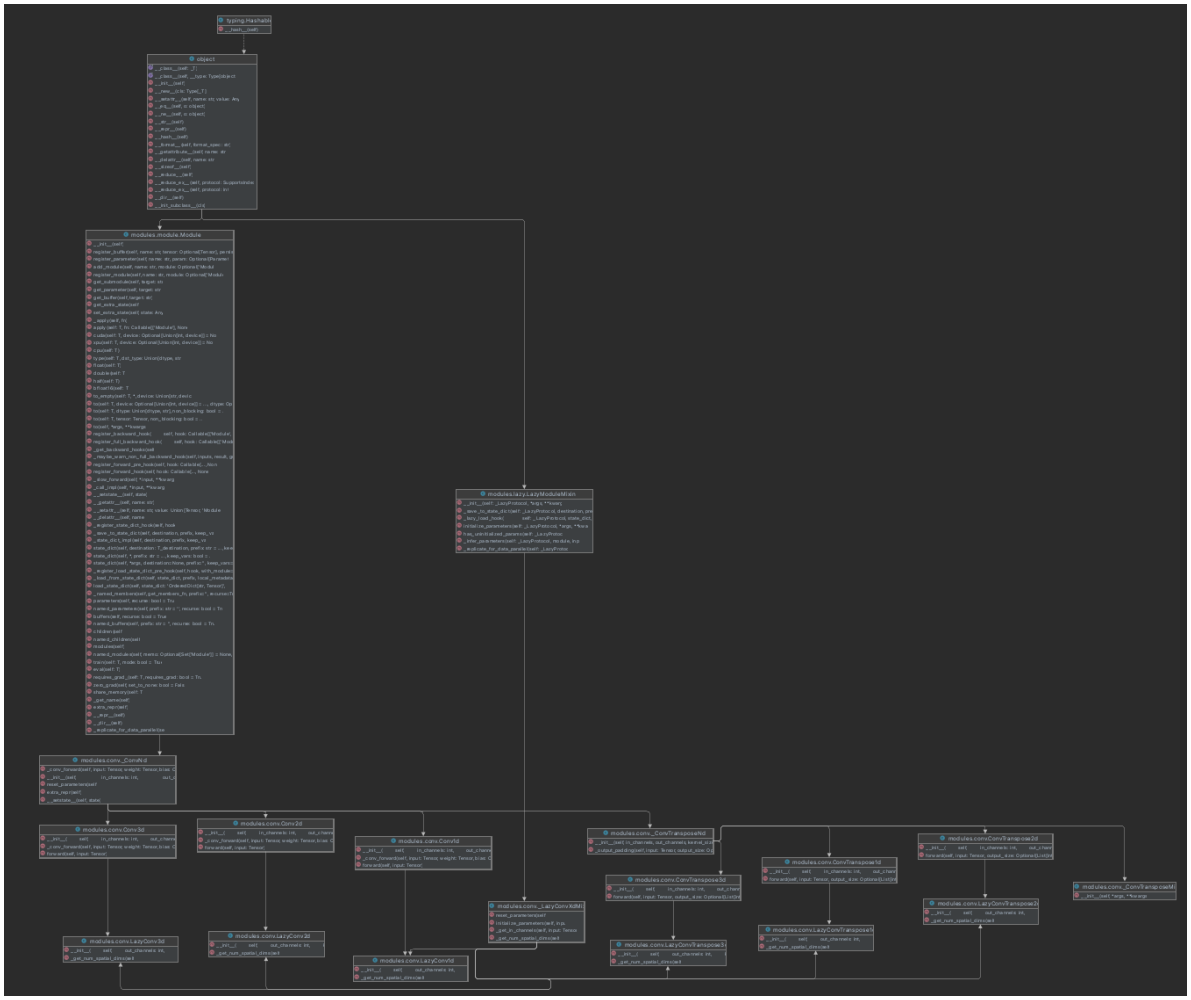

Fig 2. Class diagram

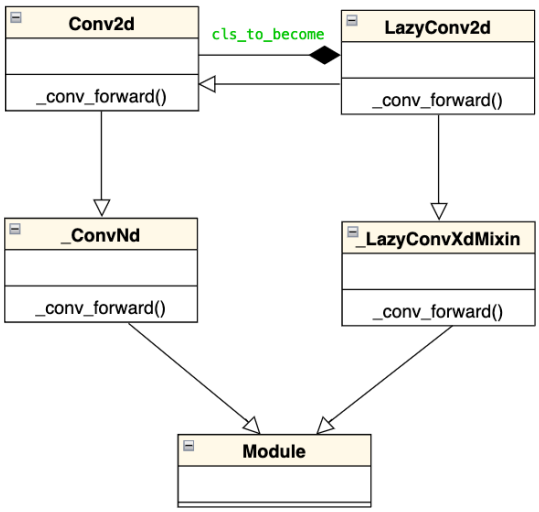

Fig 3. UML diagram indicating Bridge design pattern

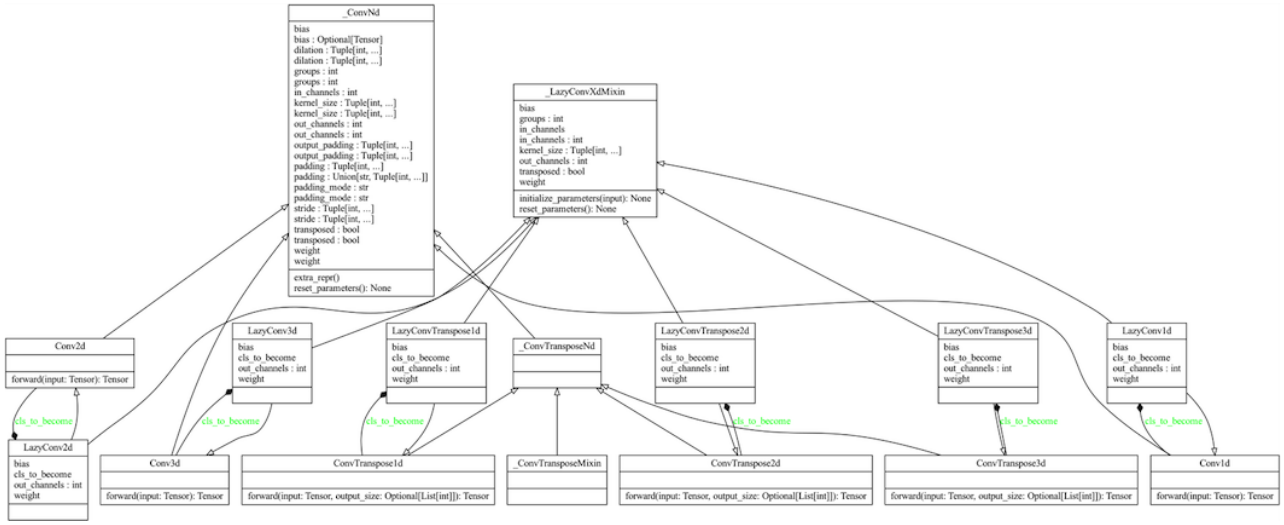

Fig 4. UML diagram generated using Pyreverse

```python
class _ConvNd(Module):

    __constants__ = ['stride', 'padding', 'dilation', 'groups',
                     'padding_mode', 'output_padding', 'in_channels',
                     'out_channels', 'kernel_size']
    __annotations__ = {'bias': Optional[torch.Tensor]}

    def _conv_forward(self, input: Tensor, weight: Tensor, bias: Optional[Tensor]) -> Tensor:
        ...
```

```python
class Conv2d(_ConvNd):
    __doc__ = r"""Applies a 2D convolution over an input signal composed of several input
    planes.
```

```python
# LazyConv2d defines weight as a Tensor but derived class defines it as UnitializeParameter
class LazyConv2d(_LazyConvXdMixin, Conv2d):  # type: ignore[misc]
    r"""A :class:`torch.nn.Conv2d` module with lazy initialization of
    the ``in_channels`` argument of the :class:`Conv2d` that is inferred from
    the ``input.size(1)``.
    The attributes that will be lazily initialized are `weight` and `bias`.

    Check the :class:`torch.nn.modules.lazy.LazyModuleMixin` for further documentation
    on lazy modules and their limitations.
```

```python
# super class define this variable as None. "type: ignore[..] is required
# since we are redefining the variable.
cls_to_become = Conv2d  # type: ignore[assignment]
```

Fig 5. Specific code

Full path - https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/conv.py#L305

B. Module: JIT

Module:jit: script module

From the simplified class diagram of the Script module, it seems to be that the Script module is following a bottom-up pattern. From my readings of week 4, this seems like a **Parameterized factory pattern/Virtual Constructor** design pattern

Factory pattern: Let the subclasses decide what the object type would be.
Uses:

- A class can't anticipate the class of objects it must create.
- A class delegates responsibility to one of the several helper subclasses, and wants to localize the knowledge of which helper subclass is the delegate.

This module follows Factory pattern because QuantizedRNNCellBase is an interface to create the objects, but its letting subclasses decide which class to implement. Pytorch essentially needs to create a QuantizedRNNCellBase object which has either one of the types QuantizedRNNCell , QuantizedGRUCell , QuantizedLSTMCell. But this decision can't be taken by the interface class at runtime, so the decision of deciding which type of object to create is left to its subclasses which are either of the 3 types mentioned above.

We also mentioned it to be a **Parameterised Factory method** because each subclass is taking an interface as its input parameter and is working on creating an object. Also we noticed that each subclass has private constructors which are inheriting the interface.

All the above mentioned reasons tick off the requirements of Factory method, which is the reason why we concluded that JIT module follows Factory method design pattern.

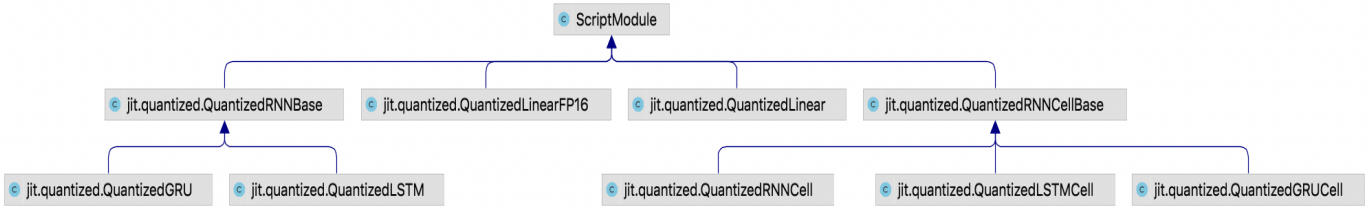

Fig 6. Simplified class diagram

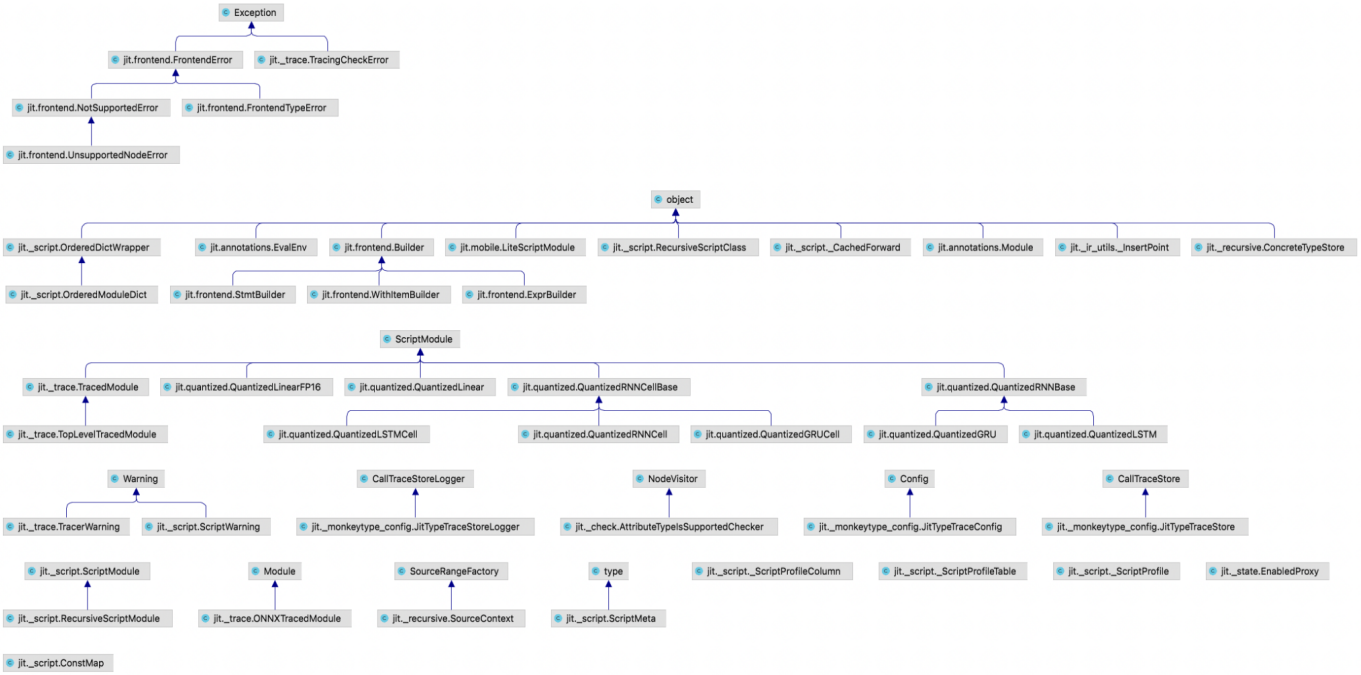

Fig 7. class diagram of entire JIT module

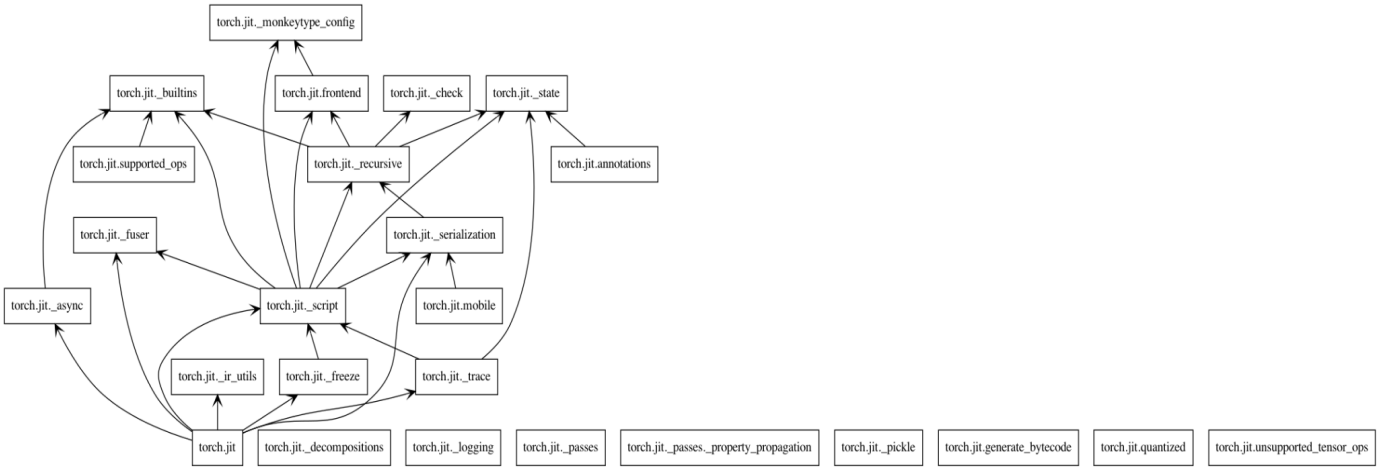

Fig 8. Simplified Class diagram for entire JIT module

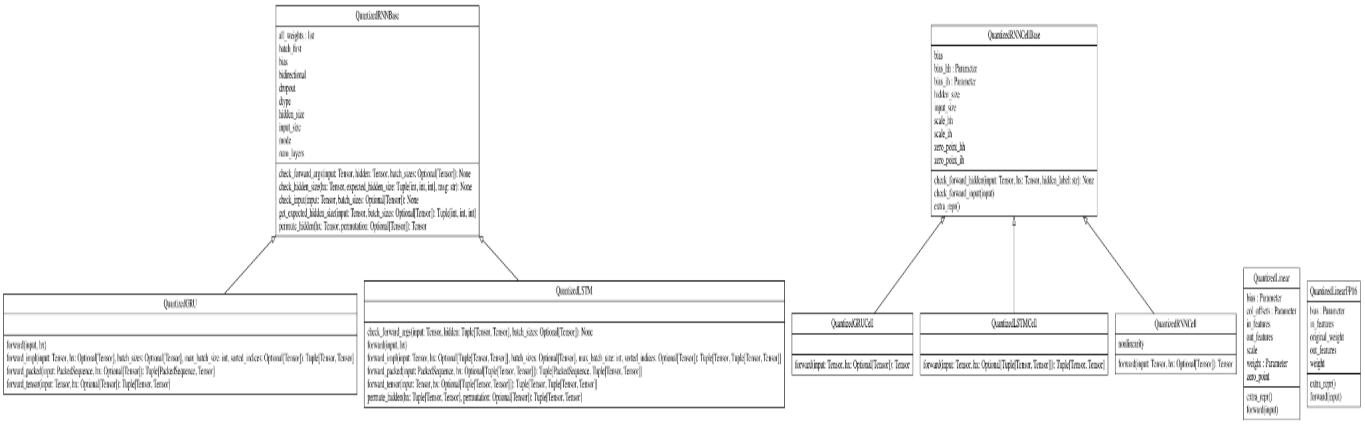

Fig 9. UML diagram for Script module

QuantisedRNNCellBase: (*double underscore in python makes the type private****)***

```
# Quantized RNN cell implementations
class QuantizedRNNCellBase(torch.jit.ScriptModule):
    __constants__ = ['input_size', 'hidden_size', 'bias', 'scale_hh', 'scale_ih',
                     'zero_point_ih', 'zero_point_hh']

    def __init__(self, other):
        super(QuantizedRNNCellBase, self).__init__()
        self.input_size = other.input_size
        self.hidden_size = other.hidden_size
        self.bias = other.bias
        if not self.bias:
            raise ValueError("Quantized RNN cells require bias terms")
```

QuantizedRNNCell under QuantizedRNNCellBase:

```
class QuantizedRNNCell(QuantizedRNNCellBase):
    __constants__ = ['input_size', 'hidden_size', 'bias', 'scale_hh', 'scale_ih',
                     'zero_point_ih', 'zero_point_hh', 'nonlinearity']

    def __init__(self, other):
        super(QuantizedRNNCell, self).__init__(other)
        self.nonlinearity = other.nonlinearity

    @torch.jit.script_method
    def forward(self, input: Tensor, hx: Optional[Tensor] = None) -> Tensor:
        self.check_forward_input(input)
```

QuantizedGRUCell under QuantizedRNNCellBase:

```
class QuantizedGRUCell(QuantizedRNNCellBase):
    def __init__(self, other):
        super(QuantizedGRUCell, self).__init__(other)
    @torch.jit.script_method
    def forward(self, input: Tensor, hx: Optional[Tensor] = None) -> Tensor:
        self.check_forward_input(input)
```

QuantizedLSTMCell under QuantizedRNNCellBase:

```
class QuantizedLSTMCell(QuantizedRNNCellBase):
    def __init__(self, other):
        super(QuantizedLSTMCell, self).__init__(other)
    @torch.jit.script_method
    def forward(self, input: Tensor, hx: Optional[Tuple[Tensor, Tensor]] = None) -> Tuple[Tensor, Tensor]:
        self.check_forward_input(input)
```

Fig 10. Code Snippets for Quantized component

Full path: https://github.com/pytorch/pytorch/blob/master/torch/jit/quantized.py

III. Security Patterns

A. Out-of-bounds read

The class *PredicateAnalyzer* checks if a predicate is needed to avoid out-of-bound accesses. Due to the way we allocate local-memory tensors, there should never be out-of-bound accesses with consumer tensors when allocated on local memory. However, accessing producer tensors still may result in out-of-bound as they are replayed as consumers. So the class *PredicateAnalyzer* in the torch module checks that parallel dimension will not generate out of bound index.

```
class PredicateAnalyzer : public OptOutDispatch {
Public:

static bool needsPredicate(TensorView* producer, TensorView* consumer) {

if (!(producer->getMemoryType() == MemoryType::Local &&
        consumer->getMemoryType() == MemoryType::Local)) {
    return true;
}
```

Link to the code snippet that handles out-of-bounds:
https://github.com/pytorch/pytorch/blob/a2802ad0b928874a130bc8631adfabc8dc3c09a7/torch/csrc/jit/codegen/cuda/lower_predicate_elimination.cpp#L54-L57

Full path to the source code: [torch/csrc/jit/codegen/cuda/lower_predicate_elimination.cpp]

IV. Implementation Plan

The proposed implementation consists of adding components to the existing Pytorch's architecture that help users to view the trained/inference models. Currently, no such support exists in PyTorch, and the users are forced to use other third-party tools from Tensorflow. However, integrating monitoring and visualization features like these would be of great use to the Pytorch developer community.
Features that are inspired by another third-party library named 'Netron' would be customized to specifically suit the needs of PyTorch models.

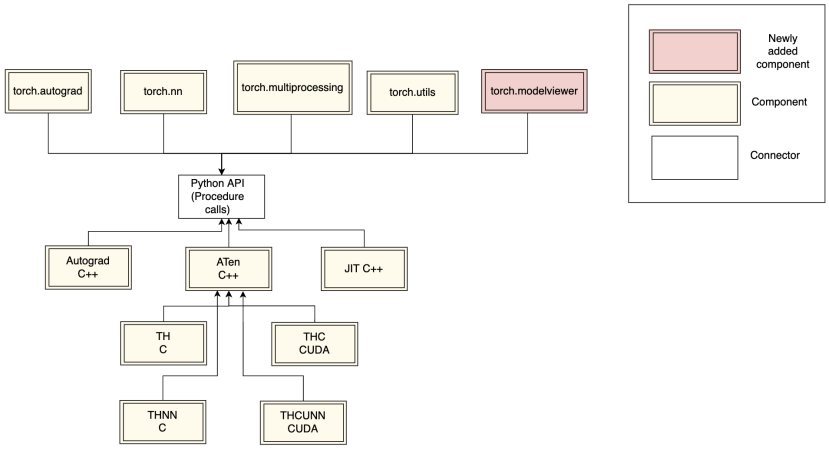

Fig 11. Implemented architecture diagram - High-level view

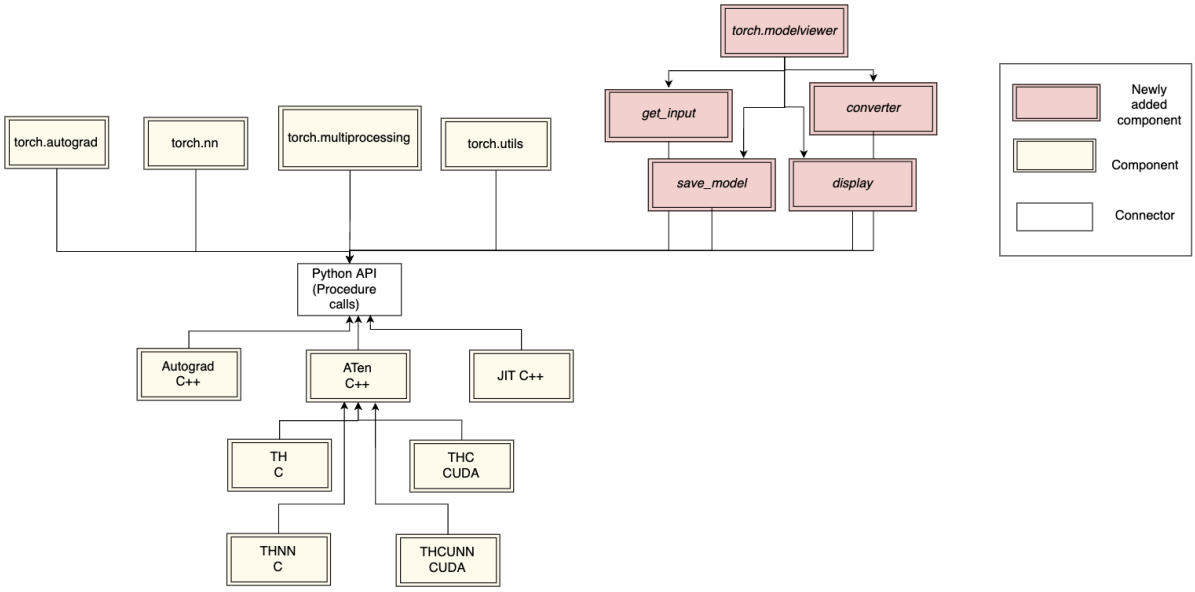

Fig 12. Implemented architecture diagram - Detailed view

Reasons why additional components do not degrade the existing architecture:

1. The additional component/module does not violate the existing principal design decisions that were initially formulated by PyTorch. Also, no new principal design decisions were added to the system's descriptive architecture that is not included in the prescriptive architecture
   - No architectural drift
2. The newly added component (torch.modelviewer) still follows the Layered architecture thereby maintaining the architecture design decision - No architectural erosion

Component detailed design:

A. Input Component

Purpose:

The purpose of this component is to have the function that gets the saved model in pytorch as the input and returns a model that can be used by other components to convert into a compatible format. A common PyTorch convention is to save models using either a .pt or .pth file extension.

Provided Interface:

Model Input -

Model Torch.modelviewer.get_input(models)

Parameters: models: The model is the actual output produced by the pyTorch after training.

Return types: input_model

Required Interface:

Model m = read_file(.pth)

Var input_model = modelviewer.get_input(m)

B. Model Converter Component

Purpose: The purpose of this component is to convert the given .pth or .pt file (Pytorch supported deep learning model) to an intermediary format that is supported by the third-party library Netron. This component will get the parameter from *Input Component,* perform a function to convert the model to be compatible with Netron, and return the netron-compatible model.

Provided Interface:

Model to Netron-compatible-model Conversion -

Model Torch.modelviewer.converter(input_model)

This method converts the input to Netron usable format.

Parameters: input_model : This parameter is returned by *get_input() function* in the input component.

Return types: netron_compatible_model:

Required Interface:

Var netron_compatible_model = modelviewer.converter(input_model)

C. Model Saver Component

Purpose:

The purpose of this component is to have the function that allows the user to save the trained model in the desired format. For example in png, jpeg. This component will take the parameter from *Model Converter Component, and* perform a function to save the model in the compatible format.

Provided Interface:

Model Saver -

Torch.modelviewer.save_model(netron_compatible_model)

Parameters: netron_compatible_model: This parameter is returned by converter() function in model converter component.

Return types: saved_netron_compatible_model

Required Interface:

Var saved_netron_compatible_model = modelviewer.save_model(netron_compatible_model)

D. Display Component

Purpose:

The purpose of this component is to have the method that will have the function (logic) to display the saved model to the user. The return type of this component will connect to the client machine to display the trained model.

Provided Interface:

Model Display -

Torch.modelviewer.display(saved_netron_compatible_model)

Parameters: *saved_netron_compatible_model:* This parameter is returned by save_model() function in the SAVER component.

Return types: aesthetically eye-pleasing UI

Required Interface:

modelviewer.display(*saved_netron_compatible_model*)

Connectors:

A. **Role: ‘**Connectors as Communicators’ which seamlessly transfers data from one method to another. It connects the components that provide monitoring and visualization features to view the trained/inference models.

‘Connectors as Converters’ to create compatible models as requested by the user.

B. **Connector type -** Procedure call

C. **Dimensions and Value**

Parameters - *Subdimension:*Data transfer
Value:model, *Name*:model
Entry point - *Subdimension*:Single
Invocation - *Subdimension:*Explicit
Value: method call
Accessibility - *Value*:Public

V. 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), PyCharm to find more of the sub-components within the main components.
- Used Pyreverse to generate UML diagrams from which design patterns can be obtained.
- 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:** ~ 25 hours

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

VI. 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. *Linkedin Learning* https://www.linkedin.com/learning/python-advanced-design-patterns
5. *Introduction to torch.nn Module.* Retrieved April 26, 2022, from https://www.educba.com/torch-dot-nn-module/
6. Quantization in Pytorch https://pytorch.org/docs/stable/quantization.html#quantization-doc
7. *PyTorch Documentation.* Retrieved April 11, 2022, from https://pytorch.org/docs/stable/
8. *PyTorch JIT and TorchScript.* Retrieved April 14, 2022, from https://towardsdatascience.com/pytorch-jit-and-torchscript-c2a77bac0fff
9. *Automatic Differentiation in PyTorch* https://openreview.net/pdf?id=BJJsrmfCZ