torch.fx

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master] [docs] [source]fx.rst, line 1)

Unknown directive type "currentmodule".

```
.. currentmodule:: torch.fx
```

Overview

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master] [docs] [source]fx.rst, line 8)

Unknown directive type "automodule".

```
.. automodule:: torch.fx
```

Writing Transformations

What is an FX transform? Essentially, it's a function that looks like this.

```
import torch
import torch.fx

def transform(m: nn.Module,
              tracer_class : type = torch.fx.Tracer) -> torch.nn.Module:
    # Step 1: Acquire a Graph representing the code in `m`

    # NOTE: torch.fx.symbolic_trace is a wrapper around a call to
    # fx.Tracer.trace and constructing a GraphModule. We'll
    # split that out in our transform to allow the caller to
    # customize tracing behavior.
    graph : torch.fx.Graph = tracer_class().trace(m)

    # Step 2: Modify this Graph or create a new one
    graph = ...

    # Step 3: Construct a Module to return
    return torch.fx.GraphModule(m, graph)
```

Your transform will take in an :class:`torch.nn.Module`, acquire a :class:`Graph` from it, do some modifications, and return a new :class:`torch.nn.Module`. You should think of the :class:`torch.nn.Module` that your FX transform returns as identical to a regular :class:`torch.nn.Module` -- you can pass it to another FX transform, you can pass it to TorchScript, or you can run it. Ensuring that the inputs and outputs of your FX transform are a :class:`torch.nn.Module` will allow for composability.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master] [docs] [source]fx.rst, line 39); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master] [docs] [source]fx.rst, line 39); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master] [docs] [source]fx.rst, line 39); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master] [docs] [source]fx.rst, line 39); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master] [docs] [source]fx.rst, line 39); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master] [docs] [source]fx.rst, line 39); *backlink*

Unknown interpreted text role "class".

Note

It is also possible to modify an existing :class:`GraphModule` instead of creating a new one, like so:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master] [docs] [source]fx.rst, line 49); *backlink*

Unknown interpreted text role "class".

```
import torch
import torch.fx

def transform(m : nn.Module) -> nn.Module:
    gm : torch.fx.GraphModule = torch.fx.symbolic_trace(m)

    # Modify gm.graph
    # <...>

    # Recompile the forward() method of `gm` from its Graph
    gm.recompile()

    return gm
```

Note that you MUST call :meth:`GraphModule.recompile` to bring the generated `forward()` method on the `GraphModule` in sync with the modified :class:`Graph`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 66); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 66); *backlink*

Unknown interpreted text role "class".

Given that you've passed in a :class:`torch.nn.Module` that has been traced into a :class:`Graph`, there are now two primary approaches you can take to building a new :class:`Graph`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 69); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 69); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 69); *backlink*

Unknown interpreted text role "class".

A Quick Primer on Graphs

Full treatment of the semantics of graphs can be found in the :class:`Graph` documentation, but we are going to cover the basics here. A :class:`Graph` is a data structure that represents a method on a :class:`GraphModule`. The information that this requires is:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 76); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 76); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 76); *backlink*

Unknown interpreted text role "class".

- What are the inputs to the method?
- What are the operations that run inside the method?
- What is the output (i.e. return) value from the method?

All three of these concepts are represented with :class:`Node` instances. Let's see what we mean by that with a short example:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 85); *backlink*

Unknown interpreted text role "class".

```
import torch
import torch.fx

class MyModule(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.param = torch.nn.Parameter(torch.rand(3, 4))
        self.linear = torch.nn.Linear(4, 5)

    def forward(self, x):
        return torch.topk(torch.sum(
            self.linear(x + self.linear.weight).relu(), dim=-1), 3)

m = MyModule()
gm = torch.fx.symbolic_trace(m)

gm.graph.print_tabular()
```

Here we define a module `MyModule` for demonstration purposes, instantiate it, symbolically trace it, then call the :meth:`Graph.print_tabular` method to print out a table showing the nodes of this :class:`Graph`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 108); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 108); *backlink*

Unknown interpreted text role "class".

opcode	name	target	args	kwargs
placeholder	x	x	()	{}
get_attr	linear_weight	linear.weight	()	{}

opcode	name	target	args	kwargs
call_function	add_1	<built-in function add>	(x, linear_weight)	{}
call_module	linear_1	linear	(add_1,)	{}
call_method	relu_1	relu	(linear_1,)	{}
call_function	sum_1	<built-in method sum ...>	(relu_1,)	{'dim': -1}
call_function	topk_1	<built-in method topk ...>	(sum_1, 3)	{}
output	output	output	(topk_1,)	{}

We can use this information to answer the questions we posed above.

- What are the inputs to the method? In FX, method inputs are specified via special `placeholder` nodes. In this case, we have a single `placeholder` node with a `target` of `x`, meaning we have a single (non-self) argument named x.
- What are the operations within the method? The `get_attr`, `call_function`, `call_module`, and `call_method` nodes represent the operations in the method. A full treatment of the semantics of all of these can be found in the :class:`Node` documentation.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 138); *backlink*
>
> Unknown interpreted text role "class".

- What is the return value of the method? The return value in a :class:`Graph` is specified by a special `output` node.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 143); *backlink*
>
> Unknown interpreted text role "class".

Given that we now know the basics of how code is represented in FX, we can now explore how we would edit a :class:`Graph`.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 146); *backlink*
>
> Unknown interpreted text role "class".

Graph Manipulation

Direct Graph Manipulation

One approach to building this new :class:`Graph` is to directly manipulate your old one. To aid in this, we can simply take the :class:`Graph` we obtain from symbolic tracing and modify it. For example, let's say we desire to replace :func:`torch.add` calls with :func:`torch.mul` calls.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 155); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 155); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 155); *backlink*
>
> Unknown interpreted text role "func".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 155); *backlink*
>
> Unknown interpreted text role "func".

```
import torch
import torch.fx

# Sample module
class M(torch.nn.Module):
    def forward(self, x, y):
        return torch.add(x, y)

def transform(m: torch.nn.Module,
              tracer_class : type = fx.Tracer) -> torch.nn.Module:
    graph : fx.Graph = tracer_class().trace(m)
    # FX represents its Graph as an ordered list of
    # nodes, so we can iterate through them.
    for node in graph.nodes:
        # Checks if we're calling a function (i.e:
        # torch.add)
        if node.op == 'call_function':
            # The target attribute is the function
            # that call_function calls.
            if node.target == torch.add:
                node.target = torch.mul

    graph.lint() # Does some checks to make sure the
                 # Graph is well-formed.

    return fx.GraphModule(m, graph)
```

We can also do more involved :class:`Graph` rewrites, such as deleting or appending nodes. To aid in these transformations, FX has utility functions for transforming the graph that can be found in the :class:`Graph` documentation. An example of using these APIs to append a :func:`torch.relu` call can be found below.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 190); *backlink*
>
> Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 190); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 190); *backlink*

Unknown interpreted text role "func".

```
# Specifies the insertion point. Any nodes added to the
# Graph within this scope will be inserted after `node`
with traced.graph.inserting_after(node):
    # Insert a new `call_function` node calling `torch.relu`
    new_node = traced.graph.call_function(
        torch.relu, args=(node,))

    # We want all places that used the value of `node` to
    # now use that value after the `relu` call we've added.
    # We use the `replace_all_uses_with` API to do this.
    node.replace_all_uses_with(new_node)
```

For simple transformations that only consist of substitutions, you can also make use of the subgraph rewriter.

Subgraph Rewriting With replace_pattern()

FX also provides another level of automation on top of direct graph manipulation. The :func:`replace_pattern` API is essentially a "find/replace" tool for editing :class:`Graph`s. It allows you to specify a `pattern` and `replacement` function and it will trace through those functions, find instances of the group of operations in the `pattern` graph, and replace those instances with copies of the `replacement` graph. This can help to greatly automate tedious graph manipulation code, which can get unwieldy as the transformations get more complex.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 217); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 217); *backlink*

Unknown interpreted text role "class".

Graph Manipulation Examples

- Replace one op
- Conv/Batch Norm fusion
- replace_pattern: Basic usage
- Quantization
- Invert Transformation

Proxy/Retracing

Another way of manipulating :class:`Graph`s is by reusing the :class:`Proxy` machinery used in symbolic tracing. For example, let's imagine that we wanted to write a transformation that decomposed PyTorch functions into smaller operations. It would transform every `F.relu(x)` call into `(x > 0) * x`. One possibility would be to perform the requisite graph rewriting to insert the comparison and multiplication after the `F.relu`, and then clean up the original `F.relu`. However, we can automate this process by using :class:`Proxy` objects to automatically record operations into the :class:`Graph`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 239); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 239); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 239); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 239); *backlink*

Unknown interpreted text role "class".

To use this method, we write the operations that we want inserted as regular PyTorch code and invoke that code with :class:`Proxy` objects as arguments. These :class:`Proxy` objects will capture the operations that are performed on them and append them to the :class:`Graph`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 249); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 249); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 249); *backlink*

Unknown interpreted text role "class".

```
# Note that this decomposition rule can be read as regular Python
def relu_decomposition(x):
```

```
    return (x > 0) * x

decomposition_rules = {}
decomposition_rules[F.relu] = relu_decomposition

def decompose(model: torch.nn.Module,
              tracer_class : type = fx.Tracer) -> torch.nn.Module:
    """
    Decompose `model` into smaller constituent operations.
    Currently,this only supports decomposing ReLU into its
    mathematical definition: (x > 0) * x
    """
    graph : fx.Graph = tracer_class().trace(model)
    new_graph = fx.Graph()
    env = {}
    tracer = torch.fx.proxy.GraphAppendingTracer(graph)
    for node in graph.nodes:
        if node.op == 'call_function' and node.target in decomposition_rules:
            # By wrapping the arguments with proxies,
            # we can dispatch to the appropriate
            # decomposition rule and implicitly add it
            # to the Graph by symbolically tracing it.
            proxy_args = [
                fx.Proxy(env[x.name], tracer) if isinstance(x, fx.Node) else x for x in node.args]
            output_proxy = decomposition_rules[node.target](*proxy_args)

            # Operations on `Proxy` always yield new `Proxy`s, and the
            # return value of our decomposition rule is no exception.
            # We need to extract the underlying `Node` from the `Proxy`
            # to use it in subsequent iterations of this transform.
            new_node = output_proxy.node
            env[node.name] = new_node
        else:
            # Default case: we don't have a decomposition rule for this
            # node, so just copy the node over into the new graph.
            new_node = new_graph.node_copy(node, lambda x: env[x.name])
            env[node.name] = new_node
    return fx.GraphModule(model, new_graph)
```

In addition to avoiding explicit graph manipulation, using :class:`Proxy`s also allows you to specify your rewrite rules as native Python code. For transformations that require a large amount of rewrite rules (such as vmap or grad), this can often improve readability and maintainability of the rules. Note that while calling :class:`Proxy` we also passed a tracer pointing to the underlying variable *graph*. This is done so if in case the operations in graph are n-ary (e.g. add is a binary operator) the call to :class:`Proxy` does not create multiple instances of a graph tracer which can lead to unexpected runtime errors. We recommend this method of using :class:`Proxy` especially when the underlying operators can not be safely assumed to be unary.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 297); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 297); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 297); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 297); *backlink*

Unknown interpreted text role "class".

A worked example of using :class:`Proxy`s for :class:`Graph` manipulation can be found here.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 309); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 309); *backlink*

Unknown interpreted text role "class".

The Interpreter Pattern

A useful code organizational pattern in FX is to loop over all the :class:`Node`s in a :class:`Graph` and execute them. This can be used for several things including runtime analysis of values flowing through the graph or transformation of the code via retracing with :class:`Proxy`s. For example, suppose we want to run a :class:`GraphModule` and record the :class:`torch.Tensor` shape and dtype properties on the nodes as we see them at runtime. That might look like:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 316); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 316); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 316); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 316); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 316); *backlink*

Unknown interpreted text role "class".

```
import torch
import torch.fx
from torch.fx.node import Node

from typing import Dict

class ShapeProp:
    """
    Shape propagation. This class takes a `GraphModule`.
    Then, its `propagate` method executes the `GraphModule`
    node-by-node with the given arguments. As each operation
    executes, the ShapeProp class stores away the shape and
    element type for the output values of each operation on
    the `shape` and `dtype` attributes of the operation's
    `Node`.
    """
    def __init__(self, mod):
        self.mod = mod
        self.graph = mod.graph
        self.modules = dict(self.mod.named_modules())

    def propagate(self, *args):
        args_iter = iter(args)
        env : Dict[str, Node] = {}

        def load_arg(a):
            return torch.fx.graph.map_arg(a, lambda n: env[n.name])

        def fetch_attr(target : str):
            target_atoms = target.split('.')
            attr_itr = self.mod
            for i, atom in enumerate(target_atoms):
                if not hasattr(attr_itr, atom):
                    raise RuntimeError(f"Node referenced nonexistant target {'.'.join(target_atoms[:i])}")
                attr_itr = getattr(attr_itr, atom)
            return attr_itr

        for node in self.graph.nodes:
            if node.op == 'placeholder':
                result = next(args_iter)
            elif node.op == 'get_attr':
                result = fetch_attr(node.target)
            elif node.op == 'call_function':
                result = node.target(*load_arg(node.args), **load_arg(node.kwargs))
            elif node.op == 'call_method':
                self_obj, *args = load_arg(node.args)
                kwargs = load_arg(node.kwargs)
                result = getattr(self_obj, node.target)(*args, **kwargs)
            elif node.op == 'call_module':
                result = self.modules[node.target](*load_arg(node.args), **load_arg(node.kwargs))

            # This is the only code specific to shape propagation.
            # you can delete this `if` branch and this becomes
            # a generic GraphModule interpreter.
            if isinstance(result, torch.Tensor):
                node.shape = result.shape
                node.dtype = result.dtype

            env[node.name] = result

        return load_arg(self.graph.result)
```

As you can see, a full interpreter for FX is not that complicated but it can be very useful. To ease using this pattern, we provide the :class:`Interpreter` class, which encompasses the above logic in a way that certain aspects of the interpreter's execution can be overridden via method overrides.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 387); *backlink*

Unknown interpreted text role "class".

In addition to executing operations, we can also generate a new *Graph* by feeding :class:`Proxy` values through an interpreter. Similarly, we provide the :class:`Transformer` class to encompass this pattern. :class:`Transformer` behaves similarly to :class:`Interpreter`, but instead of calling the `run` method to get a concrete output value from the Module, you would call the :meth:`Transformer.transform` method to return a new :class:`GraphModule` which was subject to any transformation rules you installed as overridden methods.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 393); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 393); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 393); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 393); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 393); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 393); *backlink*

Unknown interpreted text role "class".

Examples of the Interpreter Pattern

- Shape Propagation
- Performance Profiler

Debugging

Introduction

Often in the course of authoring transformations, our code will not be quite right. In this case, we may need to do some debugging. The key is to work backwards: first, check the results of invoking the generated module to prove or disprove correctness. Then, inspect and debug the generated code. Then, debug the process of transformations that led to the generated code.

If you're not familiar with debuggers, please see the auxiliary section :ref:`Available Debuggers`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 423); *backlink*

Unknown interpreted text role "ref".

Common Pitfalls in Transform Authoring

- Nondeterministic `set` iteration order. In Python, the `set` datatype is unordered. Using `set` to contain collections of objects like `Node`s, for example, can cause unexpected nondeterminism. An example is iterating over a set of `Node`s to insert them into a `Graph`. Because the `set` data type is unordered, the ordering of the operations in the output program will be nondeterministic and can change across program invocations. The recommended alternative is to use a `dict` data type, which is insertion ordered as of Python 3.7 (and as of cPython 3.6). A `dict` can be used equivalently to a set by storing values to be deduplicated in the keys of the `dict`.

Checking Correctness of Modules

Because the output of most deep learning modules consists of floating point :class:`torch.Tensor` instances, checking for equivalence between the results of two :class:`torch.nn.Module` is not as straightforward as doing a simple equality check. To motivate this, let's use an example:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 444); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 444); *backlink*

Unknown interpreted text role "class".

```
import torch
import torch.fx
import torchvision.models as models

def transform(m : torch.nn.Module) -> torch.nn.Module:
    gm = torch.fx.symbolic_trace(m)

    # Imagine we're doing some transforms here
    # <...>

    gm.recompile()

    return gm

resnet18 = models.resnet18()
transformed_resnet18 = transform(resnet18)

input_image = torch.randn(5, 3, 224, 224)

assert resnet18(input_image) == transformed_resnet18(input_image)
"""
RuntimeError: Boolean value of Tensor with more than one value is ambiguous
"""
```

Here, we've tried to check equality of the values of two deep learning models with the == equality operator. However, this is not well- defined both due to the issue of that operator returning a tensor and not a bool, but also because comparison of floating point values should use a margin of error (or epsilon) to account for the non-commutativity of floating point operations (see here for more details). We can use :func:`torch.allclose` instead, which will give us an approximate comparison taking into account a relative and absolute tolerance threshold:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 476); *backlink*

Unknown interpreted text role "func".

```
assert torch.allclose(resnet18(input_image), transformed_resnet18(input_image))
```

This is the first tool in our toolbox to check if transformed modules are behaving as we expect compared to a reference implementation.

Debugging the Generated Code

Because FX generates the `forward()` function on :class:`GraphModule`s, using traditional debugging techniques like `print` statements or `pdb` is not as straightforward. Luckily, we have several techniques we can use for debugging the generated code.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 497); *backlink*

Unknown interpreted text role "class".

Use `pdb`

Invoke `pdb` to step into the running program. Although the code that represents the :class:`Graph` is not in any source file, we can still step into it manually using `pdb` when the forward pass is invoked.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 504); backlink

Unknown interpreted text role "class".

```
import torch
import torch.fx
import torchvision.models as models

def my_pass(inp: torch.nn.Module, tracer_class : type = fx.Tracer) -> torch.nn.Module:
    graph = tracer_class().trace(inp)
    # Transformation logic here
    # <...>

    # Return new Module
    return fx.GraphModule(inp, graph)

my_module = models.resnet18()
my_module_transformed = my_pass(my_module)

input_value = torch.randn(5, 3, 224, 224)

# When this line is executed at runtime, we will be dropped into an
# interactive `pdb` prompt. We can use the `step` or `s` command to
# step into the execution of the next line
import pdb; pdb.set_trace()

my_module_transformed(input_value)
```

Print the Generated Code

If you'd like to run the same code multiple times, then it can be a bit tedious to step to the right code with `pdb`. In that case, one approach is to simply copy-paste the generated `forward` pass into your code and examine it from there.

```
# Assume that `traced` is a GraphModule that has undergone some
# number of transforms

# Copy this code for later
print(traced)
# Print the code generated from symbolic tracing. This outputs:
"""
def forward(self, y):
    x = self.x
    add_1 = x + y;  x = y = None
    return add_1
"""

# Subclass the original Module
class SubclassM(M):
    def __init__(self):
        super().__init__()

    # Paste the generated `forward` function (the one we printed and
    # copied above) here
    def forward(self, y):
        x = self.x
        add_1 = x + y;  x = y = None
        return add_1

# Create an instance of the original, untraced Module. Then, create an
# instance of the Module with the copied `forward` function. We can
# now compare the output of both the original and the traced version.
pre_trace = M()
post_trace = SubclassM()
```

Use the `to_folder` Function From `GraphModule`

:meth:`GraphModule.to_folder` is a method in `GraphModule` that allows you to dump out the generated FX code to a folder. Although copying the forward pass into the code often suffices as in :ref:`Print the Generated Code`, it may be easier to examine modules and parameters using `to_folder`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 578); backlink

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 578); backlink

Unknown interpreted text role "ref".

```
m = symbolic_trace(M())
m.to_folder("foo", "Bar")
from foo import Bar
y = Bar()
```

After running the above example, we can then look at the code within `foo/module.py` and modify it as desired (e.g. adding `print` statements or using `pdb`) to debug the generated code.

Debugging the Transformation

Now that we've identified that a transformation is creating incorrect code, it's time to debug the transformation itself. First, we'll check the :ref:`Limitations of Symbolic Tracing` section in the documentation. Once we verify that tracing is working as expected, the goal becomes figuring out what went wrong during our `GraphModule` transformation. There may be a quick answer in :ref:`Writing Transformations`, but, if not, there are several ways to examine our traced module:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 597); backlink

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 597); backlink

Unknown interpreted text role "ref".

```
# Sample Module
class M(torch.nn.Module):
    def forward(self, x, y):
        return x + y
```

```
# Create an instance of `M`
m = M()

# Symbolically trace an instance of `M` (returns a GraphModule). In
# this example, we'll only be discussing how to inspect a
# GraphModule, so we aren't showing any sample transforms for the
# sake of brevity.
traced = symbolic_trace(m)

# Print the code produced by tracing the module.
print(traced)
# The generated `forward` function is:
"""
def forward(self, x, y):
    add = x + y;  x = y = None
    return add
"""

# Print the internal Graph.
print(traced.graph)
# This print-out returns:
"""
graph():
    %x : [#users=1] = placeholder[target=x]
    %y : [#users=1] = placeholder[target=y]
    %add : [#users=1] = call_function[target=operator.add](args = (%x, %y), kwargs = {})
    return add
"""

# Print a tabular representation of the internal Graph.
traced.graph.print_tabular()
# This gives us:
"""
opcode         name    target                   args    kwargs
-------------  ------  -----------------------  ------  --------
placeholder    x       x                        ()      {}
placeholder    y       y                        ()      {}
call_function  add     <built-in function add>  (x, y)  {}
output         output  output                   (add,)  {}
"""
```

Using the utility functions above, we can compare our traced Module before and after we've applied our transformations. Sometimes, a simple visual comparison is enough to trace down a bug. If it's still not clear what's going wrong, a debugger like `pdb` can be a good next step.

Going off of the example above, consider the following code:

```
# Sample user-defined function
def transform_graph(module: torch.nn.Module, tracer_class : type = fx.Tracer) -> torch.nn.Module:
    # Get the Graph from our traced Module
    g = tracer_class().trace(module)

    """
    Transformations on `g` go here
    """

    return fx.GraphModule(module, g)

# Transform the Graph
transformed = transform_graph(traced)

# Print the new code after our transforms. Check to see if it was
# what we expected
print(transformed)
```

Using the above example, let's say that the call to `print(traced)` showed us that there was an error in our transforms. We want to find what goes wrong using a debugger. We start a `pdb` session. We can see what's happening during the transform by breaking on `transform_graph(traced)`, then pressing `s` to "step into" the call to `transform_graph(traced)`.

We may also have good luck by editing the `print_tabular` method to print different attributes of the Nodes in the Graph. (For example, we might want to see the Node's `input_nodes` and `users`.)

Available Debuggers

The most common Python debugger is pdb. You can start your program in "debug mode" with `pdb` by typing `python -m pdb FILENAME.py` into the command line, where `FILENAME` is the name of the file you want to debug. After that, you can use the `pdb` debugger commands to move through your running program stepwise. It's common to set a breakpoint (`b LINE-NUMBER`) when you start `pdb`, then call `c` to run the program until that point. This prevents you from having to step through each line of execution (using `s` or `n`) to get to the part of the code you want to examine. Alternatively, you can write `import pdb; pdb.set_trace()` before the line you want to break at. If you add `pdb.set_trace()`, your program will automatically start in debug mode when you run it. (In other words, you can just type `python FILENAME.py` into the command line instead of `python -m pdb FILENAME.py`.) Once you're running your file in debug mode, you can step through the code and examine your program's internal state using certain commands. There are many excellent tutorials on `pdb` online, including RealPython's "Python Debugging With Pdb".

IDEs like PyCharm or VSCode usually have a debugger built in. In your IDE, you can choose to either a) use `pdb` by pulling up a terminal window in your IDE (e.g. View → Terminal in VSCode), or b) use the built-in debugger (usually a graphical wrapper around `pdb`).

Limitations of Symbolic Tracing

FX uses a system of **symbolic tracing** (a.k.a symbolic execution) to capture the semantics of programs in a transformable/analyzable form. The system is **tracing** in that it executes the program (really a :class:`torch.nn.Module` or function) to record operations. It is **symbolic** in that the data flowing through the program during this execution is not real data, but rather symbols (:class:`Proxy` in FX parlance).

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 730); *backlink***
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 730); *backlink***
>
> Unknown interpreted text role "class".

Although symbolic tracing works for most neural net code, it has some limitations.

Dynamic Control Flow

The main limitation of symbolic tracing is it does not currently support *dynamic control flow*. That is, loops or `if` statements where

the condition may depend on the input values of the program.

For example, let's examine the following program:

```
def func_to_trace(x):
    if x.sum() > 0:
        return torch.relu(x)
    else:
        return torch.neg(x)

traced = torch.fx.symbolic_trace(func_to_trace)
"""
  <...>
  File "dyn.py", line 6, in func_to_trace
    if x.sum() > 0:
  File "pytorch/torch/fx/proxy.py", line 155, in __bool__
    return self.tracer.to_bool(self)
  File "pytorch/torch/fx/proxy.py", line 85, in to_bool
    raise TraceError('symbolically traced variables cannot be used as inputs to control flow')
torch.fx.proxy.TraceError: symbolically traced variables cannot be used as inputs to control flow
"""
```

The condition to the `if` statement relies on the value of `x.sum()`, which relies on the value of `x`, a function input. Since `x` can change (i.e. if you pass a new input tensor to the traced function), this is *dynamic control flow*. The traceback walks back up through your code to show you where this situation happens.

Static Control Flow

On the other hand, so-called *static control flow* is supported. Static control flow is loops or `if` statements whose value cannot change across invocations. Typically, in PyTorch programs, this control flow arises for code making decisions about a model's architecture based on hyper-parameters. As a concrete example:

```
import torch
import torch.fx

class MyModule(torch.nn.Module):
    def __init__(self, do_activation : bool = False):
        super().__init__()
        self.do_activation = do_activation
        self.linear = torch.nn.Linear(512, 512)

    def forward(self, x):
        x = self.linear(x)
        # This if-statement is so-called static control flow.
        # Its condition does not depend on any input values
        if self.do_activation:
            x = torch.relu(x)
        return x

without_activation = MyModule(do_activation=False)
with_activation = MyModule(do_activation=True)

traced_without_activation = torch.fx.symbolic_trace(without_activation)
print(traced_without_activation.code)
"""
def forward(self, x):
    linear_1 = self.linear(x);  x = None
    return linear_1
"""

traced_with_activation = torch.fx.symbolic_trace(with_activation)
print(traced_with_activation.code)
"""
import torch
def forward(self, x):
    linear_1 = self.linear(x);  x = None
    relu_1 = torch.relu(linear_1);  linear_1 = None
    return relu_1
"""
```

The if-statement `if self.do_activation` does not depend on any function inputs, thus it is static. `do_activation` can be considered to be a hyper-parameter, and the traces of different instances of `MyModule` with different values for that parameter have different code. This is a valid pattern that is supported by symbolic tracing.

Many instances of dynamic control flow are semantically static control flow. These instances can be made to support symbolic tracing by removing the data dependencies on input values, for example by moving values to `Module` attributes or by binding concrete values to arguments during symbolic tracing:

```
def f(x, flag):
    if flag: return x
    else: return x*2

fx.symbolic_trace(f) # Fails!

fx.symbolic_trace(f, concrete_args={'flag': True})
```

In the case of truly dynamic control flow, the sections of the program that contain this code can be traced as calls to the Method (see :ref`Customizing Tracing`) or function (see :func:`wrap`) rather than tracing through them.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 847); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 847); *backlink*

Unknown interpreted text role "func".

Non-`torch` Functions

FX uses `__torch_function__` as the mechanism by which it intercepts calls (see the technical overview for more information about this). Some functions, such as builtin Python functions or those in the `math` module, are not covered by `__torch_function__`, but we would still like to capture them in symbolic tracing. For example:

```
import torch
import torch.fx
from math import sqrt

def normalize(x):
    """
    Normalize `x` by the size of the batch dimension
    """
    return x / sqrt(len(x))
```

```
# It's valid Python code
normalize(torch.rand(3, 4))

traced = torch.fx.symbolic_trace(normalize)
"""
  <...>
  File "sqrt.py", line 9, in normalize
    return x / sqrt(len(x))
  File "pytorch/torch/fx/proxy.py", line 161, in __len__
    raise RuntimeError("'len' is not supported in symbolic tracing by default. If you want "
RuntimeError: 'len' is not supported in symbolic tracing by default. If you want this call to be recorded, please call torch.fx.w
"""
```

The error tells us that the built-in function `len` is not supported. We can make it so that functions like this are recorded in the trace as direct calls using the :func:`wrap` API:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 888); *backlink*
>
> Unknown interpreted text role "func".

```
torch.fx.wrap('len')
torch.fx.wrap('sqrt')

traced = torch.fx.symbolic_trace(normalize)

print(traced.code)
"""
import math
def forward(self, x):
    len_1 = len(x)
    sqrt_1 = math.sqrt(len_1);  len_1 = None
    truediv = x / sqrt_1;  x = sqrt_1 = None
    return truediv
"""
```

Customizing Tracing with the `Tracer` class

The :class:`Tracer` class is the class that underlies the implementation of `symbolic_trace`. The behavior of tracing can be customized by subclassing Tracer, like so:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 914); *backlink*
>
> Unknown interpreted text role "class".

```
class MyCustomTracer(torch.fx.Tracer):
    # Inside here you can override various methods
    # to customize tracing. See the `Tracer` API
    # reference
    pass


# Let's use this custom tracer to trace through this module
class MyModule(torch.nn.Module):
    def forward(self, x):
        return torch.relu(x) + torch.ones(3, 4)

mod = MyModule()

traced_graph = MyCustomTracer().trace(mod)
# trace() returns a Graph. Let's wrap it up in a
# GraphModule to make it runnable
traced = torch.fx.GraphModule(mod, traced_graph)
```

Leaf Modules

Leaf Modules are the modules that appear as calls in the symbolic trace rather than being traced through. The default set of leaf modules is the set of standard `torch.nn` module instances. For example:

```
class MySpecialSubmodule(torch.nn.Module):
    def forward(self, x):
        return torch.neg(x)

class MyModule(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = torch.nn.Linear(3, 4)
        self.submod = MySpecialSubmodule()

    def forward(self, x):
        return self.submod(self.linear(x))

traced = torch.fx.symbolic_trace(MyModule())
print(traced.code)
# `linear` is preserved as a call, yet `submod` is traced though.
# This is because the default set of "Leaf Modules" includes all
# standard `torch.nn` modules.
"""
import torch
def forward(self, x):
    linear_1 = self.linear(x);  x = None
    neg_1 = torch.neg(linear_1);  linear_1 = None
    return neg_1
"""
```

The set of leaf modules can be customized by overriding :meth:`Tracer.is_leaf_module`.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 974); *backlink*
>
> Unknown interpreted text role "meth".

Miscellanea

- Tensor constructors (e.g. `torch.zeros`, `torch.ones`, `torch.rand`, `torch.randn`, `torch.sparse_coo_tensor`) are currently not traceable.
    - The deterministic constructors (`zeros`, `ones`) can be used and the value they produce will be embedded in the trace as a constant. This is only problematic if the arguments to these constructors refers to dynamic input sizes. In this case, `ones_like` or `zeros_like` may be a viable substitute.
    - Nondeterministic constructors (`rand`, `randn`) will have a single random value embedded in the trace. This is likely not

the intended behavior. One workaround is to wrap `torch.randn` in a `torch.fx.wrap` function and call that instead.

```
@torch.fx.wrap
def torch_randn(x, shape):
    return torch.randn(shape)

def f(x):
    return x + torch_randn(x, 5)
fx.symbolic_trace(f)
```

- This behavior may be fixed in a future release.

- Type annotations
  - Python 3-style type annotations (e.g. `func(x : torch.Tensor, y : int) -> torch.Tensor`) are supported and will be preserved by symbolic tracing.
  - Python 2-style comment type annotations `# type: (torch.Tensor, int) -> torch.Tensor` are not currently supported.
  - Annotations on local names within a function are not currently supported.
- Gotcha around `training` flag and submodules
  - When using functionals like `torch.nn.functional.dropout`, it will be common for the training argument to be passed in as `self.training`. During FX tracing, this will likely be baked in as a constant value.

```
import torch
import torch.fx

class DropoutRepro(torch.nn.Module):
  def forward(self, x):
    return torch.nn.functional.dropout(x, training=self.training)


traced = torch.fx.symbolic_trace(DropoutRepro())
print(traced.code)
"""
def forward(self, x):
  dropout = torch.nn.functional.dropout(x, p = 0.5, training = True, inplace = False);  x = None
  return dropout
"""

traced.eval()

x = torch.randn(5, 3)
torch.testing.assert_allclose(traced(x), x)
"""
AssertionError: Tensor-likes are not close!

Mismatched elements: 15 / 15 (100.0%)
Greatest absolute difference: 1.6207983493804932 at index (0, 2) (up to 1e-05 allowed)
Greatest relative difference: 1.0 at index (0, 0) (up to 0.0001 allowed)
"""
```

  - However, when the standard `nn.Dropout()` submodule is used, the training flag is encapsulated and--because of the preservation of the `nn.Module` object model--can be changed.

```
class DropoutRepro2(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.drop = torch.nn.Dropout()

  def forward(self, x):
    return self.drop(x)

traced = torch.fx.symbolic_trace(DropoutRepro2())
print(traced.code)
"""
def forward(self, x):
  drop = self.drop(x);  x = None
  return drop
"""

traced.eval()

x = torch.randn(5, 3)
torch.testing.assert_allclose(traced(x), x)
```

  - Because of this difference, consider marking modules that interact with the `training` flag dynamically as leaf modules.

API Reference

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1082)

Unknown directive type "autofunction".

```
.. autofunction:: torch.fx.symbolic_trace
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1084)

Unknown directive type "autofunction".

```
.. autofunction:: torch.fx.wrap
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1086)

Unknown directive type "autoclass".

```
.. autoclass:: torch.fx.GraphModule
  :members:

  .. automethod:: __init__
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1091)

Unknown directive type "autoclass".

```
.. autoclass:: torch.fx.Graph
  :members:

  .. automethod:: __init__
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1096)

Unknown directive type "autoclass".

```
.. autoclass:: torch.fx.Node
  :members:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1099)

Unknown directive type "autoclass".

```
.. autoclass:: torch.fx.Tracer
  :members:
  :inherited-members:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1103)

Unknown directive type "autoclass".

```
.. autoclass:: torch.fx.Proxy
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1105)

Unknown directive type "autoclass".

```
.. autoclass:: torch.fx.Interpreter
  :members:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1108)

Unknown directive type "autoclass".

```
.. autoclass:: torch.fx.Transformer
  :members:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1111)

Unknown directive type "autofunction".

```
.. autofunction:: torch.fx.replace_pattern
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1117)

Unknown directive type "py:module".

```
.. py:module:: torch.fx.passes
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1118)

Unknown directive type "py:module".

```
.. py:module:: torch.fx.experimental
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1119)

Unknown directive type "py:module".

```
.. py:module:: torch.fx.experimental.unification
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]fx.rst, line 1120)

Unknown directive type "py:module".

```
.. py:module:: torch.fx.experimental.unification.multipledispatch
```