Ta le of Contents

TORCH.FX

Overview

This feature is ex erimental an its sta ility is not currently guarantee . Procee at your own risk

FX is a toolkit for capturing an transforming functional PyTorch programs. It consists of GraphMo ule an a correspon ing interme iate representation (IR). When GraphMo ule is constructe with an `nn.Module` instance as its argument, GraphMo ule will trace through the computation of that Mo ule's `forward` metho sym olically an recor those operations in the FX interme iate representation.

```
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)
```

The Interme iate Representation centers aroun a 5-opco e format:

```
print(gm.graph)
```

```
graph(x):
    %linear_weight : [#users=1] = self.linear.weight
    %add_1 : [#users=1] = call_function[target=<built-in function add>](args = (%x, %linear_weight), kwargs = {})
    %linear_1 : [#users=1] = call_module[target=linear](args = (%add_1,), kwargs = {})
    %relu_1 : [#users=1] = call_method[target=relu](args = (%linear_1,), kwargs = {})
    %sum_1 : [#users=1] = call_function[target=<built-in method sum of type object at 0x7ff2da9dc300>](args = (%relu_1,), kwargs = {dim:
-1})
    %topk_1 : [#users=1] = call_function[target=<built-in method topk of type object at 0x7ff2da9dc300>](args = (%sum_1, 3), kwargs = {})
    return topk_1
```

The No e semantics are as follows:

- `placeholder` represents a function input. The `name` attri ute specifies the name this value will take on. `target` is similarly the name of the argument. `args` hol s either: 1) nothing, or 2) a single argument enoting the efault parameter of the function input. `kwargs` is on't-care. Placehol ers correspon to the function parameters (e.g. `x`) in the graph printout.
- `get_attr` retrieves a parameter from the mo ule hierarchy. `name` is similarly the name the result of the fetch is assigne to. `target` is the fully-qualifie name of the parameter's position in the mo ule hierarchy. `args` an `kwargs` are on't-care
- `call_function` applies a free function to some values. `name` is similarly the name of the value to assign to. `target` is the function to e applie . `args` an `kwargs` represent the arguments to the function, following the Python calling convention
- `call_module` applies a mo ule in the mo ule hierarchy's `forward()` metho to given arguments. `name` is as previous. `target` is the fully-qualifie name of the mo ule in the mo ule hierarchy to call. `args` an `kwargs` represent the arguments to invoke the mo ule on, *including the self argument*.
- `call_method` calls a metho on a value. `name` is as similar. `target` is the string name of the metho to apply to the `self` argument. `args` an `kwargs` represent the arguments to invoke the mo ule on, *including the self argument*
- `output` contains the output of the trace function in its `args[0]` attri ute. This correspon s to the "return" statement in the Graph printout.

GraphMo ule automatically generates Python co e for the operations it sym olically o serve :

```
print(gm.code)
```

```
import torch
def forward(self, x):
    linear_weight = self.linear.weight
    add_1 = x + linear_weight
    x = linear_weight = None
    linear_1 = self.linear(add_1)
    add_1 = None
    relu_1 = linear_1.relu()
    linear_1 = None
    sum_1 = torch.sum(relu_1, dim = -1)
    relu_1 = None
    topk_1 = torch.topk(sum_1, 3)
    sum_1 = None
    return topk_1
    topk_1 = None
```

Because this co e is vali PyTorch co e, the resulting `GraphModule` can e use in any context another `nn.Module` can e use , inclu ing in TorchScript tracing/compilation.

API Reference

`torch.fx.symbolic_trace`(*root: Union[torch.nn.modules.module.Module, Callable]*) → torch.fx.graph_mo ule.GraphMo ule [SO RCE]

Sym olic tracing API

Given an `nn.Module` or function instance `root` , this function will return a `GraphModule` constructe y recor ing operations seen while tracing through `root` .

Parameters

root (*Union*[*torch.nn.Module*, *Callable*]) – Mo ule or function to e trace an converte into a Graph representation.

Returns

a Mo ule create from the recor e operations from `root` .

Return type

GraphMo ule

CLASS `torch.fx.GraphModule`(*root: Union[torch.nn.modules.module.Module, Dict[str, Any]], graph: torch.fx.graph.Graph*) [SO RCE]

GraphMo ule is an nn.Mo ule generate from an fx.Graph. Graphmo ule has a `graph` attri ute, as well as `code` an `forward` attri utes generate from that `graph` .

• WARNING

When `graph` is reassigne , `code` an `forward` will e automatically regenerate . However, if you e it the contents of the `graph` without reassigning the `graph` attri ute itself, you must call `recompile()` to up ate the generate co e.

`__init__`(*root: Union[torch.nn.modules.module.Module, Dict[str, Any]], graph: torch.fx.graph.Graph*) [SO RCE]

Construct a GraphMo ule.

Parameters

- **root** (*Union*[*torch.nn.Module*, *Dict*[*str*, *Any*]) – `root` can either e an nn.Mo ule instance or a Dict mapping strings to any attri ute type. In the case that `root` is a Mo ule, any references to Mo ule- ase o ects (via qualifie name) in the Graph's No es' `target` fiel will e copie over from the respective place within `root` 's Mo ule hierarchy into the GraphMo ule's mo ule hierarchy. In the case that `root` is a ict, the qualifie name foun in a No e's `target` will e looke up irectly in the ict's keys. The o ect mappe to y the Dict will e copie over into the appropriate place within the GraphMo ule's mo ule hierarchy.
- **gra h** (*Graph*) – `graph` contains the no es this GraphMo ule shoul use for co e generation

PROPERTY `code`

Return the Python co e generate from the `Graph` un erlying this `GraphModule` .

PROPERTY `graph`

Return the `Graph` un erlying this `GraphModule`

`recompile`() → None [SO RCE]

Recompile this GraphMo ule from its `graph` attri ute. This shoul e calle after e iting the containe `graph` , otherwise the generate co e of this `GraphModule` will e out of ate.

CLASS `torch.fx.Graph` [SO RCE]

`Graph` is the main ata structure use in the FX Interme iate Representation. It consists of a series of `Node` s, each representing callsites (or other syntactic constructs). The list of `Node` s, taken together, constitute a vali Python function.

For example, the following co e

```
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)
```

Will pro uce the following Graph:

```
print(gm.graph)
```

```
graph(x):
    %linear_weight : [#users=1] = self.linear.weight
    %add_1 : [#users=1] = call_function[target=<built-in function add>](args = (%x, %linear_weight), kwargs = {})
    %linear_1 : [#users=1] = call_module[target=linear](args = (%add_1,), kwargs = {})
    %relu_1 : [#users=1] = call_method[target=relu](args = (%linear_1,), kwargs = {})
    %sum_1 : [#users=1] = call_function[target=<built-in method sum of type object at 0x7ff2da9dc300>](args = (%relu_1,), kwargs =
{dim: -1})
    %topk_1 : [#users=1] = call_function[target=<built-in method topk of type object at 0x7ff2da9dc300>](args = (%sum_1, 3), kwargs =
{})
    return topk_1
```

The No e semantics are as follows:

- `placeholder` represents a function input. The `name` attri ute specifies the name this value will take on. `target` is similarly the name of the argument. `args` hol s either: 1) nothing, or 2) a single argument enoting the efault parameter of the function input. `kwargs` is on't-care. Placehol ers correspon to the function parameters (e.g. `x`) in the graph printout.
- `get_attr` retrieves a parameter from the mo ule hierarchy. `name` is similarly the name the result of the fetch is assigne to. `target` is the fully-qualifie name of the parameter's position in the mo ule hierarchy. `args` an `kwargs` are on't-care
- `call_function` applies a free function to some values. `name` is similarly the name of the value to assign to. `target` is the function to e applie . `args` an `kwargs` represent the arguments to the function, following the Python calling convention
- `call_module` applies a mo ule in the mo ule hierarchy's `forward()` metho to given arguments. `name` is as previous. `target` is the fully-qualifie name of the mo ule in the mo ule hierarchy to call. `args` an `kwargs` represent the arguments to invoke the mo ule on, *including the self argument*.
- `call_method` calls a metho on a value. `name` is as similar. `target` is the string name of the metho to apply to the `self` argument. `args` an `kwargs` represent the arguments to invoke the mo ule on, *including the self argument*
- `output` contains the output of the trace function in its `args[0]` attri ute. This correspon s to the "return" statement in the Graph printout.

`__init__()` [SO RCE]

Construct an empty Graph.

`call_function`(*the_function: Callable[..., Any], args: Optional[Tuple[Argument, ...]] = None, kwargs: Optional[Dict[str, Argument]] = None, type_expr: Optional[Any] = None*) → torch.fx.no e.No e [SO RCE]

Insert a `call_function` `Node` into the `Graph`. A `call_function` no e represents a call to a Python calla le, specifie y `the_function`. `the_function` can e

Parameters

- **the_function** (*Callable*[.., *Any*]) – The function to e calle . Can e any PyTorch operator, Python function, or mem er of the `builtins` or `operator` namespaces.
- **args** (*Optional*[*Tuple*[*Argument*, ..]]) – The positional arguments to e passe to the calle function.
- **kwargs** (*Optional*[*Dict*[*str*, *Argument*]]) – The keywor arguments to e passe to the calle function
- **ty e_ex r** (*Optional*[*Any*]) – an optional type annotation representing the Python type the output of this no e will have.

Returns

The newly create an inserte `call_function` no e.

• NOTE

The same insertion point an type expression rules apply for this metho as `Graph.create_node()`.

`call_method`(*method_name: str, args: Optional[Tuple[Argument, ...]] = None, kwargs: Optional[Dict[str, Argument]] = None, type_expr: Optional[Any] = None*) → torch.fx.no e.No e [SO RCE]

Insert a `call_method` `Node` into the `Graph`. A `call_method` no e represents a call to a given metho on the 0th element of `args.

Parameters

- **metho _name** (*str*) – The name of the metho to apply to the self argument. For example, if args[0] is a `Node` representing a `Tensor`, then to call `relu()` on that `Tensor`, pass `relu` to `method_name`.
- **args** (*Optional*[*Tuple*[*Argument*, ..]]) – The positional arguments to e passe to the calle metho . Note that this *should* inclu e a `self` argument.
- **kwargs** (*Optional*[*Dict*[*str*, *Argument*]]) – The keywor arguments to e passe to the calle metho
- **ty e_ex r** (*Optional*[*Any*]) – an optional type annotation representing the Python type the output of this no e will have.

Returns

The newly create an inserte `call_method` no e.

• NOTE

The same insertion point an type expression rules apply for this metho as `Graph.create_node()`.

`call_module`(*module_name: str*, *args: Optional[Tuple[Argument, ...]] = None*, *kwargs: Optional[Dict[str, Argument]] = None*, *type_expr: Optional[Any] = None*) → torch.fx.no e.No e [SO RCE]

Insert a `call_module` `Node` into the `Graph`. A `call_module` no e represents a call to the forwar () function of a `Module` in the `Module` hierarchy.

Parameters

- **mo ule_name** (*str*) – The qualifie name of the `Module` in the `Module` hierarchy to e calle . For example, if the trace `Module` has a su mo ule name `foo`, which has a su mo ule name `bar`, the qualifie name `foo.bar` shoul e passe as `module_name` to call that mo ule.
- **args** (*Optional*[*Tuple*[*Argument*, ..]]) – The positional arguments to e passe to the calle metho . Note that this shoul *not* inclu e a `self` argument.
- **kwargs** (*Optional*[*Dict*[*str*, *Argument*]]) – The keywor arguments to e passe to the calle metho
- **ty e_ex r** (*Optional*[*Any*]) – an optional type annotation representing the Python type the output of this no e will have.

Returns

The newly-create an inserte `call_module` no e.

• NOTE

The same insertion point an type expression rules apply for this metho as `Graph.create_node()`.

`create_node`(*op: str*, *target: Union[Callable[[...], Any], str]*, *args: Optional[Tuple[Argument, ...]] = None*, *kwargs: Optional[Dict[str, Argument]] = None*, *name: Optional[str] = None*, *type_expr: Optional[Any] = None*) → torch.fx.no e.No e [SO RCE]

Create a `Node` an a it to the `Graph` at the current insert-point. Note that the current insert-point can e set via `Graph.inserting_before()` an `Graph.inserting_after()`.

Parameters

- **o** (*str*) – the opco e for this No e. One of 'call_function', 'call_metho ', 'get_attr', 'call_mo ule', 'placehol er', or 'output'. The semantics of these opco es are escri e in the `Graph` ocstring.
- **args** (*Optional*[*Tuple*[*Argument*, ..]]) – is a tuple of arguments to this no e.
- **kwargs** (*Optional*[*Dict*[*str*, *Argument*]]) – the kwargs of this No e
- **name** (*Optional*[*str*]) – an optional string name for the `Node`. This will influence the name of the value assigne to in the Python generate co e.
- **ty e_ex r** (*Optional*[*Any*]) – an optional type annotation representing the Python type the output of this no e will have.

Returns

The newly-create an inserte no e.

`erase_node`(*to_erase: torch.fx.node.Node*) → None [SO RCE]

Erases a `Node` from the `Graph`. Throws an exception if there are still users of that no e in the `Graph`.

Parameters

to_erase (*Node*) – The `Node` to erase from the `Graph`.

`get_attr`(*qualified_name: str*, *type_expr: Optional[Any] = None*) → torch.fx.no e.No e [SO RCE]

Insert a `get_attr` no e into the Graph. A `get_attr` `Node` represents the fetch of an attri ute from the `Module` hierarchy.

Parameters

- **ualifie _name** (*str*) – the fully-qualifie name of the attri ute to e retrieve . For example, if the trace Mo ule has a su mo ule name `foo`, which has a su mo ule name `bar`, which has an attri ute name `baz`, the qualifie name `foo.bar.baz` shoul e passe as `qualified_name`.
- **ty e_ex r** (*Optional*[*Any*]) – an optional type annotation representing the Python type the output of this no e will have.

Returns

The newly-create an inserte `get_attr` no e.

• NOTE

The same insertion point an type expression rules apply for this metho as `Graph.create_node`.

`graph_copy(g: torch.fx.graph.Graph, val_map: Dict[torch.fx.node.Node, torch.fx.node.Node])` → nion[Tuple[Any, ...], List[Any], Dict[str, Any], slice, No e, str, int, float, ool, torch. type, torch.Tensor, None] [SO RCE]

Copy all no es from a given graph into `self`.

Parameters

- **g** (*Graph*) – The source graph from which to copy No es.
- **val_ma** (*Dict*[*Node*, *Node*]) – a ictionary that will e populate with a mapping from no es in `g` to no es in `self`. Note that `val_map` can e passe in with values in it alrea y to overri e copying of certain values.

Returns

The value in `self` that is now equivalent to the output value in `g`, if `g` ha an `output` no e. `None` otherwise.

`inserting_after(n: Optional[torch.fx.node.Node] = None)` [SO RCE]

Set the point at which create_no e an companion metho s will insert into the graph. When use within a 'with' statement, this will temporary set the insert point an then restore it when the with statement exits:

```
with g.inserting_after(n):
    ... # inserting after node n
... # insert point restored to what it was previously
g.inserting_after(n) #  set the insert point permanently
```

Parameters

n (*Optional*[*Node*]) – The no e efore which to insert. If None this will insert after the eginning of the entire graph.

Returns

A resource manager that will restore the insert point on `__exit__`.

`inserting_before(n: Optional[torch.fx.node.Node] = None)` [SO RCE]

Set the point at which create_no e an companion metho s will insert into the graph. When use within a 'with' statement, this will temporary set the insert point an then restore it when the with statement exits:

```
with g.inserting_before(n):
    ... # inserting before node n
... # insert point restored to what it was previously
g.inserting_before(n) #  set the insert point permanently
```

Parameters

n (*Optional*[*Node*]) – The no e efore which to insert. If None this will insert efore the eginning of the entire graph.

Returns

A resource manager that will restore the insert point on `__exit__`.

`lint(root: Optional[torch.nn.modules.module.Module] = None)` [SO RCE]

Runs various checks on this Graph to make sure it is well-forme . In particular:

- Checks No es have correct ownership (owne y this graph)
- Checks No es appear in topological or er
- If `root` is provi e , checks that targets exist in `root`

Parameters

root (*Optional*[*torch.nn.Module*]) – The root mo ule with which to check for targets. This is equivalent to the `root` argument that is passe when constructing a `GraphModule`.

`node_copy(node: torch.fx.node.Node, arg_transform: Callable[[torch.fx.node.Node], Argument] = <function Graph.<lambda>>)` → torch.fx.no e.No e [SO RCE]

Copy a no e from one graph into another. `arg_transform` nee s to transform arguments from the graph of no e to the graph of self. Example:

```
# Copying all the nodes in `g` into `new_graph`
g : torch.fx.Graph = ...
new_graph = torch.fx.graph()
value_remap = {}
for node in g.nodes:
    value_remap[node] = new_graph.node_copy(node, lambda n : value_remap[n])
```

Parameters

- **no e** (*Node*) – The no e to copy into `self`.
- **arg_transform** (*Callable*[[*Node*], *Argument*]) – A function that transforms `Node` arguments in no e's `args` an `kwargs` into the equivalent argument in `self`. In the simplest case, this shoul retrieve a value out of a ta le mapping No es in the original graph to `self`.

PROPERTY `nodes`

Get the list of No es that constitute this Graph.

Note that this `Node` list representation is a ou ly-linke list. Mutations uring iteration (e.g. ete a No e, a a No e) are safe.

Returns

A ou ly-linke list of No es. Note that `reversed` can e calle on this list to switch iteration or er.

`output`(*result: Union[Tuple[Any, ...], List[Any], Dict[str, Any], slice, Node, str, int, float, bool, torch.dtype, torch.Tensor, None], type_expr: Optional[Any] = None*) [SO RCE]

Insert an `output` `Node` into the `Graph`. An `output` no e represents a `return` statement in Python co e. `result` is the value that shoul e returne .

Parameters

- **result** (*Argument*) – The value to e returne .
- **ty e_ex r** (*Optional*[*Any*]) – an optional type annotation representing the Python type the output of this no e will have.

• NOTE

The same insertion point an type expression rules apply for this metho as `Graph.create_node`.

`placeholder`(*name: str*, *type_expr: Optional[Any] = None*) → torch.fx.no e.No e [SO RCE]

Insert a `placeholder` no e into the Graph. A `placeholder` represents a function input.

Parameters

- **name** (*str*) – A name for the input value. This correspon s to the name of the positional argument to the function this `Graph` represents.
- **ty e_ex r** (*Optional*[*Any*]) – an optional type annotation representing the Python type the output of this no e will have. This is nee e in some cases for proper co e generation (e.g. when the function is use su sequently in TorchScript compilation).

• NOTE

The same insertion point an type expression rules apply for this metho as `Graph.create_node`.

`python_code`(*root_module: str*) → str [SO RCE]

Turn this `Graph` into vali Python co e.

Parameters

root_mo ule (*str*) – The name of the root mo ule on which to look-up qualifie name targets. This is usually 'self'.

Returns

The string source co e generate from this `Graph`.

CLASS `torch.fx.Node`(*graph: Graph, name: str, op: str, target: Union[Callable[[...], Any], str], args: Tuple[Union[Tuple[Any, ...], List[Any], Dict[str, Any], slice, Node, str, int, float, bool, torch.dtype, torch.Tensor, None], ...], kwargs: Dict[str, Union[Tuple[Any, ...], List[Any], Dict[str, Any], slice, Node, str, int, float, bool, torch.dtype, torch.Tensor, None]], type: Optional[Any] = None*) [SO RCE]

PROPERTY `all_input_nodes`

Return all No es that are inputs to this No e. This is equivalent to iterating over `args` an `kwargs` an only collecting the values that are No es

`append`(*x: torch.fx.node.Node*) [SO RCE]

Insert x after this no e in the list of no es in the graph. Equvalent to `self.next.prepend(x)`

Parameters

x (*Node*) – The no e to put after this no e. Must e a mem er of the same graph.

PROPERTY `args`

Return the tuple of arguments to this No e. The interpretation of arguments epen s on the no e's opco e. See the `fx.Graph` ocstring for more information.

PROPERTY `kwargs`

Return the ict of kwargs to this No e. The interpretation of arguments epen s on the no e's opco e. See the `fx.Graph` ocstring for more information.

PROPERTY `next`

Get the next no e in the linke list

`prepend(x: torch.fx.node.Node)` [SO RCE]

Insert x efore this no e in the list of no es in the graph.

Before: p -> self

x -> x -> ax

After: p -> x -> self

x -> ax

Parameters

x (*Node*) – The no e to put efore this no e. Must e a mem er of the same graph.

PROPERTY `prev`

Get the previous no e in the linke list

`replace_all_uses_with(replace_with: torch.fx.node.Node)` → List[torch.fx.no e.No e] [SO RCE]

Replace all uses of `self` in the Graph with the No e `replace_with`. Returns the list of no es on which this change was ma e.

CLASS `torch.fx.Tracer` [SO RCE]

`Tracer` is the class that implements the sym olic tracing functionality of `torch.fx.symbolic_trace`. A call to `symbolic_trace(m)` is equivalent to `Tracer().trace(m)`.

Tracer can e su classe to overri e various ehaviors of the tracing process. The ifferent ehaviors that can e overri en are escri e in the ocstrings of the metho s on this class.

`call_module(m: torch.nn.modules.module.Module, forward: Callable[[...], Any], args, kwargs)` [SO RCE]

Metho that specifies the ehavior of this `Tracer` when it encounters a call to an `nn.Module` instance.

By efault, the ehavior is to check if the calle mo ule is a leaf mo ule via `is_leaf_module`. If it is, emit a `call_module` no e referring to `m` in the `Graph`. Otherwise, call the `Module` normally, tracing through the operations in its `forward` function.

This metho can e overri en to–for example–create neste trace GraphMo ules, or any other ehavior you woul want while tracing across `Module` oun aries.

`create_arg(a: Any)` → nion[Tuple[Any, ...], List[Any], Dict[str, Any], slice, No e, str, int, float, ool, torch. type, torch.Tensor, None] [SO RCE]

A metho to specify the ehavior of tracing when preparing values to e use as arguments to no es in the `Graph`.

By efault, the ehavior inclu es: - Iterate through collection types (e.g. tuple, list, ict) an recursively

call `create_args` on the elements.

- Given a Proxy o ect, return a reference to the un erlying IR `Node`
- Given a non-Proxy Tensor o ect, emit IR for various cases:
  - For a Parameter, emit a `get_attr` no e referring to that Parameter
  - For a non-Parameter Tensor, store the Tensor away in a special attri ute referring to that attri ute.

This metho can e overri en to support more types.

`create_args_for_root(root_fn, is_module)` [SO RCE]

Create `placeholder` no es correspon ing to the signature of the `root` Mo ule. This metho introspects `root`'s signature an emits those no es accor ingly, also supporting *args an **kwargs.

`is_leaf_module(m: torch.nn.modules.module.Module, module_qualified_name: str)` → ool [SO RCE]

A metho to specify whether a given `nn.Module` is a "leaf" mo ule.

Leaf mo ules are the atomic units that appear in the IR, reference y `call_module` calls. By efault, Mo ules in the PyTorch stan ar li rary namespace (torch.nn) are leaf mo ules. All other mo ules are trace through an their constituent ops are recor e , unless specifie otherwise via this parameter.

Args m: The mo ule itself mo ule_qualifie _name: The path to root of this mo ule. For example

Args m - The mo ule itself mo ule_qualifie _name - The path to root of this mo ule. For example,

> if you have a mo ule hierarchy where su mo ule `foo` contains su mo ule `bar`, which contains su mo ule `baz`, that mo ule will appear with the qualifie name `foo.bar.baz` here.

`path_of_module`(*mod*) → str [SO RCE]

Helper metho to fin the qualifie name of `mod` in the Mo ule hierarchy of `root`. For example, if `root` has a su mo ule name `foo`, which has a su mo ule name `bar`, passing `bar` into this function will return the string "foo. ar".

`trace`(*root: Union[torch.nn.modules.module.Module, Callable]*) → torch.fx.graph.Graph [SO RCE]

Trace `root` an return the correspon ing FX `Graph` representation. `root` can either e an `nn.Module` instance or a Python calla le.

‹ Previous Next ›