torchrecorder Documentation

Release 1.0.3

Gautham Venkatasubramanian

Sep 04, 2022

CONTENTS

`torchrecorder` is a Python package that can be used to record the execution graph of a `torch.nn.Module` and use it to render a visualization of the network structure via `graphviz`.

Licensed under MIT License.

CHAPTER

ONE

INSTALLATION

Requirements:

- Python3.6+
- PyTorch v1.3 or greater (the `cpu` version)
- The Graphviz library and `graphviz` python package

Install via `pip`:

```
$ pip install torchrecorder
```

CHAPTER

TWO

SIMPLE EXAMPLE

```python
import sys
import torch
import torchrecorder


class SampleNet(torch.nn.Module):
    def __init__(self):
        torch.nn.Module.__init__(self)

        self.linear_1 = torch.nn.Linear(in_features=3, out_features=3, bias=True)
        self.linear_2 = torch.nn.Linear(in_features=3, out_features=3, bias=True)
        self.linear_3 = torch.nn.Linear(in_features=6, out_features=1, bias=True)
        self.my_special_relu = torch.nn.ReLU()

    def forward(self, inputs):
        x = self.linear_1(inputs)
        y = self.linear_2(inputs)
        z = torch.cat([x, y], dim=1)
        z = self.my_special_relu(self.linear_3(z))
        return z


def main():
    i = int(sys.argv[1])
    net = SampleNet()
    torchrecorder.render_network(
        net,
        name="Sample Net",
        input_shapes=(1, 3),
        directory="./",
        fmt="svg",
        render_depth=i,
    )
```

`render_depth` = 1	`render_depth` = 2
Net-1.svg	Net-2.svg

CHAPTER

THREE

CONTENTS

3.1 User Guide

`torchrecorder` is pure Python3 code, it does not contain any C modules.

3.1.1 Installation

Requirements:

- PyTorch v1.3 or greater (the `cpu` version is only required)
- The Graphviz library and its Python interface

Install via `pip` and PyPI:

```
$ pip install torchrecorder
```

Install via `pip` and the Github repo:

```
$ pip install git+https://github.com/ahgamut/torchrecorder/
```

3.1.2 Examples

The default usage is via the *`render_network`* wrapper function.

```
import sys
import torch
import torchrecorder


class SampleNet(torch.nn.Module):
    def __init__(self):
        torch.nn.Module.__init__(self)

        self.linear_1 = torch.nn.Linear(in_features=3, out_features=3, bias=True)
        self.linear_2 = torch.nn.Linear(in_features=3, out_features=3, bias=True)
        self.linear_3 = torch.nn.Linear(in_features=6, out_features=1, bias=True)
        self.my_special_relu = torch.nn.ReLU()

    def forward(self, inputs):
        x = self.linear_1(inputs)
        y = self.linear_2(inputs)
```

(continues on next page)

(continued from previous page)

```
        z = torch.cat([x, y], dim=1)
        z = self.my_special_relu(self.linear_3(z))
        return z


def main():
    i = int(sys.argv[1])
    net = SampleNet()
    torchrecorder.render_network(
        net,
        name="Sample Net",
        input_shapes=(1, 3),
        directory="./",
        fmt="svg",
        render_depth=i,
    )
```

Net-1.svg

The render_network function calls *record* and *make_dot* so the call to *render_network* in the above example could be written as below, to allow for any modifications to the `Digraph` after rendering.

```
def main2():
    i = int(sys.argv[1])
    net = SampleNet()
    # equivalent to calling render_network
    rec = torchrecorder.record(net, name="Sample Net", input_shapes=(1, 3))
    g = torchrecorder.make_dot(rec, render_depth=i)
    # g is graphviz.Digraph object
    g.format = "svg"
    g.attr(label="{} at depth={}".format("Sample Net", i))
    g.render("{}-{}".format("Sample Net", i), directory="./", cleanup=True)
```

Styling `graphviz` attributes

To change the default styling attributes of every node, you can pass any number of `graphviz`-related attributes[1] as keyword arguments to *render_network* (or *make_dot*). The below example sets `Lato` as the default font.

```
def main():
    net = SampleNet()
    rec = torchrecorder.record(net, name="Sample Net", input_shapes=(1, 3))
    g = torchrecorder.make_dot(rec, render_depth=1, fontname="Lato")
    g.format = "svg"
    g.attr(label="Font Change via styler_args")
    g.render("{}-{}".format("StyleArgs", 1), directory="./", cleanup=True)
```

[1] the list of graphviz node attributes can be seen at https://graphviz.gitlab.io/_pages/doc/info/attrs.html

Custom Styler Objects

If the default styling of node shapes/colors is not sufficient, you can create a subclass of *GraphvizStyler* and pass it to *make_dot* via the `styler_cls` argument. The subclass needs to accepts `graphviz` attributes as keyword arguments, and override the *style_node* and *style_edge* methods.

In the below example, I construct a styler subclass that shows some parameters of `Conv2d` objects, draws orange edges out of `Conv2d` objects, and blue edges into `ReLU` objects:

```
class ConvSample(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = torch.nn.Conv2d(
            in_channels=1, out_channels=5, kernel_size=5, stride=2, padding=2
        )
        self.conv2 = torch.nn.Conv2d(
            in_channels=5, out_channels=5, kernel_size=3, stride=1, padding=1
        )
        self.relu = torch.nn.ReLU(inplace=False)

    def forward(self, x):
        x1 = self.conv1(x)
        x2 = self.conv2(x1)
        x3 = x1 + x2
        return self.relu(x3)

class MyStyler(GraphvizStyler):
    def style_node(self, node):
        default = super().style_node(node)
        if isinstance(node.fn, torch.nn.Conv2d):
            params = {}
            params["kernel_size"] = node.fn.kernel_size
            params["padding"] = node.fn.padding
            params["stride"] = node.fn.stride
            default["label"] = (
                node.name
                + "\n("
                + ",\n".join("{}={}".format(k, v) for k, v in params.items())
                + ")"
            )
            default["penwidth"] = "2.4"
        return default

    def style_edge(self, fnode, tnode):
        if isinstance(fnode.fn, torch.nn.Conv2d) and isinstance(tnode.fn, torch.
↪Tensor):
            return {"penwidth": "4.8", "color": "#ee8800"}
        elif isinstance(tnode.fn, torch.nn.ReLU) and isinstance(fnode.fn, torch.
↪Tensor):
            return {"penwidth": "4.8", "color": "#00228f"}
        else:
            return super().style_edge(fnode, tnode)


def main():
    net = ConvSample()
    rec = torchrecorder.record(net, name="ConvSample", input_shapes=(1, 1, 10, 10))
```

(continues on next page)

(continued from previous page)

```
    g = torchrecorder.make_dot(rec, render_depth=1, styler_cls=MyStyler, fontname=
↪"Lato")
    g.format = "svg"
    g.attr(label="Custom Styler Class")
    g.render("{}-{}".format("CustomStyler", 1), directory="./", cleanup=True)
```

Rendering into different formats

Currently, the `torchrecorder` package only provides rendering into `graphviz` objects, but the rendering functionality can be extended by subclassing the *`BaseRenderer`* class in a manner similar to the *`GraphvizRenderer`*. You can read the source code to see how the subclassing can be done.

If you create a subclass of *`BaseRenderer`* for a new rendering format, submit a pull request! I've been trying to render in a SigmaJS-compatible format, but haven't been able to.

3.2 API Reference

3.2.1 Convenience Functions

`torchrecorder.`**`render_network`**(*net*, *name*, *input_shapes*, *directory*, *filename=None*, *fmt='svg'*, *input_data=None*, *render_depth=1*, ***styler_args*)

Render the structure of a `torch.nn.Module` to an image via `graphviz`.

Parameters

- **`net`** (`torch.nn.Module`) –
- **`name`** (*`str`*) – name of the network
- **`input_shapes`** (*`None, tuple or list(tuple)`*) – `tuple` if `net` has a single input, `list` (`tuple`), `None` if `input_data` is provided
- **`directory`** (*`str`*) – directory to store the rendered image
- **`fmt`** (*`str, optional`*) – image format
- **`input_data`** (`torch.Tensor` or `tuple` (`torch.Tensor`), optional) – if `net` requires normalized inputs, provide them here instead of setting `input_shapes`.
- **`render_depth`** (*`int, optional`*) – Default `1`.
- **`**styler_args`** – node attributes to pass to `graphviz`

`torchrecorder.`**`record`**(*net*, *name*, *input_shapes*, *input_data=None*)

Record the graph by running a single pass of a `torch.nn.Module`.

Parameters

- **`net`** (`torch.nn.Module`) –
- **`name`** (*`str`*) – name of the network
- **`input_shapes`** (*`None, tuple or list(tuple)`*) – `tuple` if `net` has a single input, `list` (`tuple`), `None` if `input_data` is provided

- **input_data** (torch.Tensor or tuple (torch.Tensor), optional) – if net requires normalized inputs, provide them here instead of setting input_shapes.

Returns a *Recorder* object containing the execution graph

torchrecorder.**make_dot**(*rec*, *render_depth=256*, *styler_cls=None*, ***styler_args*)

Produces Graphviz representation from a *Recorder* object

Parameters

- **rec** (*Recorder*) –
- **render_depth** (*int*) – depth until which nodes should be rendered
- **styler_cls** – styler class to instantiate when styling nodes. If None, defaults to *GraphvizStyler*.

Kwargs: styler_args (optional): styler properties to be set for all nodes

Returns a graphviz.Digraph with the rendered nodes

Custom graphviz styling

class torchrecorder.renderer.**GraphvizStyler**(***styler_args*)

Bases: object

Provide styling options before rendering to graphviz.

styles

contains style properties for each subclass of *BaseNode*

Type dict

style_node(*node*)

Construct style properties for the given node.

Can be overridden to perform custom styling.

Parameters **node** (*BaseNode*) –

Returns a dict containing the required style properties

style_edge(*fnode*, *tnode*)

Construct style properties to render the given edge

Parameters

- **fnode** – *BaseNode*
- **tnode** – *BaseNode*

Returns a dict containing the required style properties

The *style_node* and *style_edge* methods read the properties *BaseNode* objects, so any subclass of *GraphvizStyler* would need the same.

class torchrecorder.nodes.**TensorNode**(*name=''*, *fn=None*, *depth=-1*, *parent=None*)

Bases: *torchrecorder.nodes.BaseNode*

Node to encapsulate a torch.Tensor.

fn

Type torch.Tensor

name
name of the *fn*

Type str

depth
int, scope depth of *fn*

Type int

parent
a *fn* in whose scope the current *fn* exists

Type object

class torchrecorder.nodes.**OpNode**(*name=''*, *fn=None*, *depth=-1*, *parent=None*)
Bases: *torchrecorder.nodes.BaseNode*

Node to encapsulate an Op, a `grad_fn` attribute of a `torch.Tensor`.

fn

Type `torch.Tensor`

name
name of the *fn*

Type str

depth
int, scope depth of *fn*

Type int

parent
a `Module` in whose `forward` the current *OpNode.fn* was executed

Type object

class torchrecorder.nodes.**LayerNode**(*name=''*, *fn=None*, *depth=-1*, *parent=None*)
Bases: *torchrecorder.nodes.BaseNode*

Node to encapsulate a `torch.nn.Module`.

fn

Type `torch.nn.Module`

name
name of the *fn*

Type str

depth
int, scope depth of *fn*

Type int

parent
a `Module` in whose `forward` *fn* was called

Type object

subnets
a set `Module` s or `grad_fn` s which are called in *fn* 's `forward`

Type set

pre
handle to the prehook on *fn*

post
handle to the hook on *fn*

class torchrecorder.nodes.**ParamNode**(*name=''*, *fn=None*, *depth=-1*, *parent=None*)
Bases: *torchrecorder.nodes.TensorNode*

Node to encapsulate a `torch.nn.Parameter`.

fn

Type `torch.nn.Parameter`

name
name of the *fn*

Type str

depth
`int`, scope depth of *fn*

Type int

parent
a `Module` whose `parameters` contains *fn*

Type object

class torchrecorder.nodes.**BaseNode**(*name=''*, *fn=None*, *depth=-1*, *parent=None*)
Bases: `object`

Wrapper object to encapsulate recorded information.

fn
an `object` recorded by the *Recorder*

Type object

name
name of the *fn*

Type str

depth
`int`, scope depth of *fn*

Type int

parent
a *fn* in whose scope the current *fn* exists

Type object

3.2.2 Custom Rendering

If you are creating a new format to render information from a *Recorder*, you would need to subclass the following methods in *BaseRenderer*, as done in *GraphvizRenderer*:

- *render_node*
- *render_recursive_node*
- *render_edge*

class torchrecorder.renderer.**GraphvizRenderer**(*rec*, *render_depth=256*, *styler_cls=None*, ***styler_args*)

Bases: *torchrecorder.renderer.base.BaseRenderer*

Render information from a *Recorder* into a graphviz.Digraph.

styler

GraphvizStyler or a subclass

Type class

render_node(*g*, *node*)

Render a node in graphviz

Renders node into the Digraph g, after applying appropriate styling. If node is a *LayerNode*, checks *render_depth* to see if its *subnets* have to rendered.

Parameters

- **g** (graphviz.Digraph) –
- **node** (*BaseNode*) –

render_recursive_node(*g*, *node*)

Render a *LayerNode* and its subnets.

Parameters

- **g** (graphviz.Digraph) –
- **node** (*LayerNode*) – has a *depth* greater than *render_depth*

The node is rendered as a separate Digraph and then is added as a graphviz.Digraph.subgraph to g.

render_edge(*g*, *fnode*, *tnode*)

Render an edge in graphviz

Parameters

- **g** (graphviz.Digraph) –
- **fnode** (*BaseNode*) –
- **tnode** (*BaseNode*) –

class torchrecorder.renderer.base.**BaseRenderer**(*rec*, *render_depth=256*)

Bases: object

Base Class for rendering information from a *Recorder*.

rec

Type *Recorder*

render_depth

nodes having a greater depth than this value will not be rendered

Type int

processed
An `OrderedDict` whose keys contain `nodes` and values contain the corresponding (directed) edge lists

Type `collections.OrderedDict`

3.2.3 Custom Recording

Subclassing *Recorder* should be unnecessary in most cases.

class torchrecorder.recorder.**Recorder**
Bases: `object`

Record and store execution graph information

fn_set
a set of objects (*fn*s) that contain recordable information

Type set

nodes
a mapping of *fn*s to their corresponding *BaseNode*s

Type dict

fn_types
a count of *fn*s by type for naming

Type dict

edges
a set of edges, each a pair of *fn*s

Type set(tuple)

add_node(*net*, *depth=0*, *parent=None*, *name=None*)
Construct a node of recording graph.

Construct a *BaseNode* that will store information related to `net` as the neural network is run.

Parameters

- **net** – Object whose information will be stored as the `fn` attribute of a *BaseNode*
- **depth** – The scope depth at which `net` is found
- **parent** – The object as part of which net will be run
- **name** – a name to recognize the object during rendering, defaults to class name

Returns `None`

add_dummy(*dummy*, *fn*)
Point to an existing node to assist recording.

Instead of creating a separate node, the `dummy` object is used to point to an existing node containing `fn`. Used for dummy ops and `AccumulateGradients` (see *leaf_dummy*).

Parameters

- **dummy** – a dummy `torch.Tensor` or op that should not be recorded
- **fn** – a recorded object that will be connected to further ops

add_edge(*_from*, *_to*)
Construct an edge of the recording graph.

Records an edge between two *fn* objects to be used while rendering. This will be used along with the `nodes` dictionary to map edges properly.

Parameters

- **_from** (*fn*) –
- **_to** (*fn*) –

register_hooks(*net*, *depth=0*, *parent=None*, *name=None*)
Register the hooks of the *Recorder* recursively on a `torch.nn.Module`.

The hooks registered are `partial` versions of *prehook* and *posthook* corresponding to each node.

Parameters

- **net** (`Module`) –
- **depth** (*int*) –
- **parent** (`torch.nn.Module`) – the parent of `net`
- **name** (*str*) – name of `net`

Returns `None`

remove_hooks()
Remove hooks from any `Module`s in *LayerNode*s.

After the recording is completed, the hooks in *LayerNode*s are unnecessary. They are removed to prevent any possible issues.

`torchrecorder.recorder.`**op_acc**(*gf*, *rec*, *node*)
Operator Accumulator.

Creates an *OpNode* to record the newly-performed operation `gf`, if not already recorded. If `gf` is an initialization op (`AccumulateGradient`), then points `gf` to its connected `torch.Tensor` instead of creating an *OpNode*. Otherwise recursively checks all operations that are connected to `gf` and adds them if necessary.

Parameters

- **gf** – current operation, a `grad_fn` object obtained from a `torch.Tensor`
- **rec** – a *Recorder* object whose nodes are updated
- **node** – *LayerNode* whose `fn` the current operation is a part of

Returns `None`

`torchrecorder.recorder.`**tensor_acc**(*tensor*, *rec*, *node*)
Tensor Accumulator.

Creates a *TensorNode* to record the newly-created tensor, if not already recorded. Note that the resulting *TensorNode* has the same parent as `node`, because the `tensor` is the output of/input to `node.fn`.

Parameters

- **tensor** – a `torch.Tensor`
- **rec** – a *Recorder* object whose nodes are updated
- **node** – a *LayerNode* whose `fn` outputs/inputs `tensor`

Returns `None`

torchrecorder.recorder.**param_acc**(*param*, *rec*, *node*)
Parameter Accumulator.

Creates a *ParamNode* to record the parameter `param` of `node.fn`, if not already recorded. Note that `node.fn` is the *parent* of `param`.

Parameters

- **param** – a `Parameter`
- **rec** – the *Recorder* object whose nodes are updated
- **node** – *LayerNode* whose `fn` contains `param`

Returns `None`

torchrecorder.recorder.**leaf_dummy**(*tensor*, *rec*)
Performs a dummy operation (adding 0) to a leaf `Tensor`.

This ensures that the (possibly in-place) operations performed on `tensor` hereafter can be correctly mapped. The dummy tensor (and operation) are not recorded separately, they merely point to the original tensor.

Parameters

- **tensor** – a newly-formed leaf `torch.Tensor`
- **rec** – the *Recorder* object whose nodes are updated

Returns `tensor` after adding 0

torchrecorder.recorder.**prehook**(*module*, *inputs*, *rec*, *node*)
hook to record BEFORE the given `module` is run.

Records parameters contained in `module`, then checks each tensor in `inputs` for any operations that may have run after the end of the previous `module`. The `inputs` are then converted to leaf tensors and recorded before being passed off to the `module`.

Parameters

- **module** – a `torch.nn.Module`
- **inputs** – a `torch.Tensor` or a `tuple` of `torch.Tensor`s
- **rec** – a *Recorder* object for global information
- **node** (*LayerNode*) – `node.fn` is `module`.

Returns the `leaf`-equivalent of `inputs`.

torchrecorder.recorder.**posthook**(*module*, *inputs*, *outputs*, *rec*, *node*)
hook to record AFTER the given `module` has run and returned.

Records any operations that may have run as part of `module`, then checks if each tensor in the `outputs` has already been recorded by a sub`module` of the current `module` (the sub`module`'s *posthook* would execute first!). If necessary, the `outputs` are converted to leaf tensors to record operations afresh.

Parameters

- **module** – a `torch.nn.Module`
- **inputs** – a `torch.Tensor` or a tuple of `torch.Tensor`s
- **outputs** – a `torch.Tensor` or a tuple of `torch.Tensor`s
- **rec** – a *Recorder* object for global information
- **node** (*LayerNode*) – `node.fn` is `module`.

Returns the `leaf`-equivalent of `outputs`.

3.3 License

MIT License

Copyright (c) 2019-2020 Gautham Venkatasubramanian

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

INDEX

S

T