TorchScript

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 4)

Unknown directive type "toctree".

```
.. toctree::
   :maxdepth: 1
   :caption: Builtin Functions
   :hidden:

   torch.jit.supported_ops <jit_builtin_functions>
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 12)

Unknown directive type "toctree".

```
.. toctree::
    :maxdepth: 1
    :caption: Language Reference
    :hidden:

    jit_language_reference
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 20)

Unknown directive type "toctree".

```
.. toctree::
    :maxdepth: 1

    jit_language_reference_v2
```


- Creating TorchScript Code
- Mixing Tracing and Scripting
- TorchScript Language
- Built-in Functions and Modules
  - PyTorch Functions and Modules
  - Python Functions and Modules
  - Python Language Reference Comparison
- Debugging
  - Disable JIT for Debugging
  - Inspecting Code
  - Interpreting Graphs
  - Tracer
- Frequently Asked Questions
- Known Issues
- Appendix
  - Migrating to PyTorch 1.2 Recursive Scripting API
  - References

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 29)

Unknown directive type "automodule".

```
.. automodule:: torch.jit
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 30)

Unknown directive type "currentmodule".

```
.. currentmodule:: torch.jit
```

TorchScript is a way to create serializable and optimizable models from PyTorch code. Any TorchScript program can be saved from a Python process and loaded in a process where there is no Python dependency.

We provide tools to incrementally transition a model from a pure Python program to a TorchScript program that can be run independently from Python, such as in a standalone C++ program. This makes it possible to train models in PyTorch using familiar tools in Python and then export the model via TorchScript to a production environment where Python programs may be disadvantageous for performance and multi-threading reasons.

For a gentle introduction to TorchScript, see the Introduction to TorchScript tutorial.

For an end-to-end example of converting a PyTorch model to TorchScript and running it in C++, see the Loading a PyTorch Model in C++ tutorial.

Creating TorchScript Code

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 50)

Unknown directive type "autosummary".

```
.. autosummary::
    :toctree: generated
    :nosignatures:

    script
    trace
    script_if_tracing
    trace_module
    fork
    wait
    ScriptModule
    ScriptFunction
    freeze
    optimize_for_inference
    set_fusion_strategy
    save
    load
    ignore
    unused
    isinstance
```

```
    Attribute
    annotate
```

Mixing Tracing and Scripting

In many cases either tracing or scripting is an easier approach for converting a model to TorchScript. Tracing and scripting can be composed to suit the particular requirements of a part of a model.

Scripted functions can call traced functions. This is particularly useful when you need to use control-flow around a simple feed-forward model. For instance the beam search of a sequence to sequence model will typically be written in script but can call an encoder module generated using tracing.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 86)

Unknown directive type "testsetup".

```
.. testsetup::

    # These are hidden from the docs, but these are necessary for `doctest`
    # since the `inspect` module doesn't play nicely with the execution
    # environment for `doctest`
    import torch

    original_script = torch.jit.script
    def script_wrapper(obj, *args, **kwargs):
        obj.__module__ = 'FakeMod'
        return original_script(obj, *args, **kwargs)

    torch.jit.script = script_wrapper

    original_trace = torch.jit.trace
    def trace_wrapper(obj, *args, **kwargs):
        obj.__module__ = 'FakeMod'
        return original_trace(obj, *args, **kwargs)

    torch.jit.trace = trace_wrapper
```

Example (calling a traced function in script):

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 110)

Unknown directive type "testcode".

```
.. testcode::

    import torch

    def foo(x, y):
        return 2 * x + y

    traced_foo = torch.jit.trace(foo, (torch.rand(3), torch.rand(3)))

    @torch.jit.script
    def bar(x):
        return traced_foo(x, x)
```

Traced functions can call script functions. This is useful when a small part of a model requires some control-flow even though most of the model is just a feed-forward network. Control-flow inside of a script function called by a traced function is preserved correctly.

Example (calling a script function in a traced function):

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 130)

Unknown directive type "testcode".

```
.. testcode::

    import torch

    @torch.jit.script
    def foo(x, y):
        if x.max() > y.max():
            r = x
        else:
            r = y
        return r


    def bar(x, y, z):
        return foo(x, y) + z

    traced_bar = torch.jit.trace(bar, (torch.rand(3), torch.rand(3), torch.rand(3)))
```

This composition also works for `nn.Module`s as well, where it can be used to generate a submodule using tracing that can be called from the methods of a script module.

Example (using a traced module):

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 153)

Unknown directive type "testcode".

```
.. testcode::
    :skipif: torchvision is None

    import torch
    import torchvision

    class MyScriptModule(torch.nn.Module):
        def __init__(self):
            super(MyScriptModule, self).__init__()
            self.means = torch.nn.Parameter(torch.tensor([103.939, 116.779, 123.68])
                                            .resize_(1, 3, 1, 1))
            self.resnet = torch.jit.trace(torchvision.models.resnet18(),
                                          torch.rand(1, 3, 224, 224))

        def forward(self, input):
            return self.resnet(input - self.means)

    my_script_module = torch.jit.script(MyScriptModule())
```

TorchScript Language

TorchScript is a statically typed subset of Python, so many Python features apply directly to TorchScript. See the full :ref:`language-reference` for details.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 176); backlink

Unknown interpreted text role "ref".

Built-in Functions and Modules

TorchScript supports the use of most PyTorch functions and many Python built-ins. See :ref:`builtin-functions` for a full reference of supported functions.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 185); backlink

Unknown interpreted text role "ref".

PyTorch Functions and Modules

TorchScript supports a subset of the tensor and neural network functions that PyTorch provides. Most methods on Tensor as well as functions in the `torch` namespace, all functions in `torch.nn.functional` and most modules from `torch.nn` are supported in TorchScript.

See :ref:`jit_unsupported` for a list of unsupported PyTorch functions and modules.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 196); backlink

Unknown interpreted text role "ref".

Python Functions and Modules

Many of Python's built-in functions are supported in TorchScript. The :any:`math` module is also supported (see :ref:`math-module` for details), but no other Python modules (built-in or third party) are supported.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 201); backlink

Unknown interpreted text role "any".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 201); backlink

Unknown interpreted text role "ref".

Python Language Reference Comparison

For a full listing of supported Python features, see :ref:`python-language-reference`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 209); backlink

Unknown interpreted text role "ref".

Debugging

Disable JIT for Debugging

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 218)

Unknown directive type "envvar".

```
.. envvar:: PYTORCH_JIT
```

Setting the environment variable `PYTORCH_JIT=0` will disable all script and tracing annotations. If there is hard-to-debug error in one of your TorchScript models, you can use this flag to force everything to run using native Python. Since TorchScript (scripting and tracing) is disabled with this flag, you can use tools like `pdb` to debug the model code. For example:

```
@torch.jit.script
def scripted_fn(x : torch.Tensor):
    for i in range(12):
        x = x + x
    return x

def fn(x):
    x = torch.neg(x)
    import pdb; pdb.set_trace()
    return scripted_fn(x)

traced_fn = torch.jit.trace(fn, (torch.rand(4, 5),))
traced_fn(torch.rand(3, 4))
```

Debugging this script with `pdb` works except for when we invoke the :func:`@torch.jit.script <torch.jit.script>` function. We can globally disable JIT, so that we can call the :func:`@torch.jit.script <torch.jit.script>` function as a normal Python function and not compile it. If the above script is called `disable_jit_example.py`, we can invoke it like so:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 240); backlink

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 240); backlink

Unknown interpreted text role "func".

```
$ PYTORCH_JIT=0 python disable_jit_example.py
```

and we will be able to step into the :func:`@torch.jit.script <torch.jit.script>` function as a normal Python function. To disable the TorchScript compiler for a specific function, see :func:`@torch.jit.ignore <torch.jit.ignore>`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 248); backlink

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 248); *backlink*

Unknown interpreted text role "func".

Inspecting Code

TorchScript provides a code pretty-printer for all :class:`ScriptModule` instances. This pretty-printer gives an interpretation of the script method's code as valid Python syntax. For example:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 258); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 262)

Unknown directive type "testcode".

```
.. testcode::

    @torch.jit.script
    def foo(len):
        # type: (int) -> torch.Tensor
        rv = torch.zeros(3, 4)
        for i in range(len):
            if i < 10:
                rv = rv - 1.0
            else:
                rv = rv + 1.0
        return rv

    print(foo.code)
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 277)

Unknown directive type "testoutput".

```
.. testoutput::
    :hide:

    ...
```

A :class:`ScriptModule` with a single `forward` method will have an attribute `code`, which you can use to inspect the :class:`ScriptModule`'s code. If the :class:`ScriptModule` has more than one method, you will need to access `.code` on the method itself and not the module. We can inspect the code of a method named `foo` on a :class:`ScriptModule` by accessing `.foo.code`. The example above produces this output:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 282); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 282); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 282); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 282); *backlink*

Unknown interpreted text role "class".

```
def foo(len: int) -> Tensor:
    rv = torch.zeros([3, 4], dtype=None, layout=None, device=None, pin_memory=None)
    rv0 = rv
    for i in range(len):
        if torch.lt(i, 10):
            rv1 = torch.sub(rv0, 1., 1)
        else:
            rv1 = torch.add(rv0, 1., 1)
        rv0 = rv1
    return rv0
```

This is TorchScript's compilation of the code for the `forward` method. You can use this to ensure TorchScript (tracing or scripting) has captured your model code correctly.

Interpreting Graphs

TorchScript also has a representation at a lower level than the code pretty- printer, in the form of IR graphs.

TorchScript uses a static single assignment (SSA) intermediate representation (IR) to represent computation. The instructions in this format consist of ATen (the C++ backend of PyTorch) operators and other primitive operators, including control flow operators for loops and conditionals. As an example:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 317)

Unknown directive type "testcode".

```
.. testcode::

    @torch.jit.script
    def foo(len):
        # type: (int) -> torch.Tensor
        rv = torch.zeros(3, 4)
        for i in range(len):
            if i < 10:
                rv = rv - 1.0
            else:
                rv = rv + 1.0
        return rv

    print(foo.graph)
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-

master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 332)

Unknown directive type "testoutput".

```
.. testoutput::
    :hide:

    ...
```

`graph` follows the same rules described in the :ref:`inspecting-code` section with regard to `forward` method lookup.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 337); *backlink*

Unknown interpreted text role "ref".

The example script above produces the graph:

```
graph(%len.1 : int):
  %24 : int = prim::Constant[value=1]()
  %17 : bool = prim::Constant[value=1]() # test.py:10:5
  %12 : bool? = prim::Constant()
  %10 : Device? = prim::Constant()
  %6 : int? = prim::Constant()
  %1 : int = prim::Constant[value=3]() # test.py:9:22
  %2 : int = prim::Constant[value=4]() # test.py:9:25
  %20 : int = prim::Constant[value=10]() # test.py:11:16
  %23 : float = prim::Constant[value=1]() # test.py:12:23
  %4 : int[] = prim::ListConstruct(%1, %2)
  %rv.1 : Tensor = aten::zeros(%4, %6, %6, %10, %12) # test.py:9:10
  %rv : Tensor = prim::Loop(%len.1, %17, %rv.1) # test.py:10:5
    block0(%i.1 : int, %rv.14 : Tensor):
      %21 : bool = aten::lt(%i.1, %20) # test.py:11:12
      %rv.13 : Tensor = prim::If(%21) # test.py:11:9
        block0():
          %rv.3 : Tensor = aten::sub(%rv.14, %23, %24) # test.py:12:18
          -> (%rv.3)
        block1():
          %rv.6 : Tensor = aten::add(%rv.14, %23, %24) # test.py:14:18
          -> (%rv.6)
      -> (%17, %rv.13)
  return (%rv)
```

Take the instruction `%rv.1 : Tensor = aten::zeros(%4, %6, %6, %10, %12) # test.py:9:10` for example.

- `%rv.1 : Tensor` means we assign the output to a (unique) value named `rv.1`, that value is of `Tensor` type and that we do not know its concrete shape.
- `aten::zeros` is the operator (equivalent to `torch.zeros`) and the input list `(%4, %6, %6, %10, %12)` specifies which values in scope should be passed as inputs. The schema for built-in functions like `aten::zeros` can be found at Builtin Functions.
- `# test.py:9:10` is the location in the original source file that generated this instruction. In this case, it is a file named *test.py*, on line 9, and at character 10.

Notice that operators can also have associated `blocks`, namely the `prim::Loop` and `prim::If` operators. In the graph print-out, these operators are formatted to reflect their equivalent source code forms to facilitate easy debugging.

Graphs can be inspected as shown to confirm that the computation described by a :class:`ScriptModule` is correct, in both automated and manual fashion, as described below.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 380); *backlink*

Unknown interpreted text role "class".

Tracer

Tracing Edge Cases

There are some edge cases that exist where the trace of a given Python function/module will not be representative of the underlying code. These cases can include:

- Tracing of control flow that is dependent on inputs (e.g. tensor shapes)
- Tracing of in-place operations of tensor views (e.g. indexing on the left-hand side of an assignment)

Note that these cases may in fact be traceable in the future.

Automatic Trace Checking

One way to automatically catch many errors in traces is by using `check_inputs` on the `torch.jit.trace()` API. `check_inputs` takes a list of tuples of inputs that will be used to re-trace the computation and verify the results. For example:

```
def loop_in_traced_fn(x):
    result = x[0]
    for i in range(x.size(0)):
        result = result * x[i]
    return result

inputs = (torch.rand(3, 4, 5),)
check_inputs = [(torch.rand(4, 5, 6),), (torch.rand(2, 3, 4),)]

traced = torch.jit.trace(loop_in_traced_fn, inputs, check_inputs=check_inputs)
```

Gives us the following diagnostic information:

```
ERROR: Graphs differed across invocations!
Graph diff:

            graph(%x : Tensor) {
            %1 : int = prim::Constant[value=0]()
            %2 : int = prim::Constant[value=0]()
            %result.1 : Tensor = aten::select(%x, %1, %2)
            %4 : int = prim::Constant[value=0]()
            %5 : int = prim::Constant[value=0]()
            %6 : Tensor = aten::select(%x, %4, %5)
            %result.2 : Tensor = aten::mul(%result.1, %6)
            %8 : int = prim::Constant[value=0]()
            %9 : int = prim::Constant[value=1]()
            %10 : Tensor = aten::select(%x, %8, %9)
        -   %result : Tensor = aten::mul(%result.2, %10)
        +   %result.3 : Tensor = aten::mul(%result.2, %10)
        ?          ++
            %12 : int = prim::Constant[value=0]()
            %13 : int = prim::Constant[value=2]()
            %14 : Tensor = aten::select(%x, %12, %13)
        +   %result : Tensor = aten::mul(%result.3, %14)
        +   %16 : int = prim::Constant[value=0]()
        +   %17 : int = prim::Constant[value=3]()
        +   %18 : Tensor = aten::select(%x, %16, %17)
        -   %15 : Tensor = aten::mul(%result, %14)
        ?     ^                                ^
        +   %19 : Tensor = aten::mul(%result, %18)
        ?     ^                                ^
        -   return (%15);
        ?             ^
```

```
+   return (%19);
?            ^
    }
```

This message indicates to us that the computation differed between when we first traced it and when we traced it with the `check_inputs`. Indeed, the loop within the body of `loop_in_traced_fn` depends on the shape of the input `x`, and thus when we try another `x` with a different shape, the trace differs.

In this case, data-dependent control flow like this can be captured using :func:`torch.jit.script` instead:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 461); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 464)

Unknown directive type "testcode".

```
.. testcode::

    def fn(x):
        result = x[0]
        for i in range(x.size(0)):
            result = result * x[i]
        return result

    inputs = (torch.rand(3, 4, 5),)
    check_inputs = [(torch.rand(4, 5, 6),), (torch.rand(2, 3, 4),)]

    scripted_fn = torch.jit.script(fn)
    print(scripted_fn.graph)
    #print(str(scripted_fn.graph).strip())

    for input_tuple in [inputs] + check_inputs:
        torch.testing.assert_close(fn(*input_tuple), scripted_fn(*input_tuple))
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 482)

Unknown directive type "testoutput".

```
.. testoutput::
    :hide:

    ...
```

Which produces:

```
graph(%x : Tensor) {
    %5 : bool = prim::Constant[value=1]()
    %1 : int = prim::Constant[value=0]()
    %result.1 : Tensor = aten::select(%x, %1, %1)
    %4 : int = aten::size(%x, %1)
    %result : Tensor = prim::Loop(%4, %5, %result.1)
    block0(%i : int, %7 : Tensor) {
        %10 : Tensor = aten::select(%x, %1, %i)
        %result.2 : Tensor = aten::mul(%7, %10)
        -> (%5, %result.2)
    }
    return (%result);
}
```

Tracer Warnings

The tracer produces warnings for several problematic patterns in traced computation. As an example, take a trace of a function that contains an in-place assignment on a slice (a view) of a Tensor:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 510)

Unknown directive type "testcode".

```
.. testcode::

    def fill_row_zero(x):
        x[0] = torch.rand(*x.shape[1:2])
        return x

    traced = torch.jit.trace(fill_row_zero, (torch.rand(3, 4),))
    print(traced.graph)
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 519)

Unknown directive type "testoutput".

```
.. testoutput::
    :hide:

    ...
```

Produces several warnings and a graph which simply returns the input:

```
fill_row_zero.py:4: TracerWarning: There are 2 live references to the data region being modified when tracing in-place operator copy_ (po
    x[0] = torch.rand(*x.shape[1:2])
fill_row_zero.py:6: TracerWarning: Output nr 1. of the traced function does not match the corresponding output of the Python function. De
Not within tolerance rtol=1e-05 atol=1e-05 at input[0, 1] (0.09115803241729736 vs. 0.6782537698745728) and 3 other locations (33.00%)
    traced = torch.jit.trace(fill_row_zero, (torch.rand(3, 4),))
graph(%0 : Float(3, 4)) {
    return (%0);
}
```

We can fix this by modifying the code to not use the in-place update, but rather build up the result tensor out-of-place with `torch.cat`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 538)

Unknown directive type "testcode".

```
.. testcode::

    def fill_row_zero(x):
        x = torch.cat((torch.rand(1, *x.shape[1:2]), x[1:2]), dim=0)
        return x
```

```
traced = torch.jit.trace(fill_row_zero, (torch.rand(3, 4),))
print(traced.graph)
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 547)

Unknown directive type "testoutput".

```
.. testoutput::
    :hide:

    ...
```

Frequently Asked Questions

Q: I would like to train a model on GPU and do inference on CPU. What are the best practices?

First convert your model from GPU to CPU and then save it, like so:

```
cpu_model = gpu_model.cpu()
sample_input_cpu = sample_input_gpu.cpu()
traced_cpu = torch.jit.trace(cpu_model, sample_input_cpu)
torch.jit.save(traced_cpu, "cpu.pt")

traced_gpu = torch.jit.trace(gpu_model, sample_input_gpu)
torch.jit.save(traced_gpu, "gpu.pt")

# ... later, when using the model:

if use_gpu:
  model = torch.jit.load("gpu.pt")
else:
  model = torch.jit.load("cpu.pt")

model(input)
```

This is recommended because the tracer may witness tensor creation on a specific device, so casting an already-loaded model may have unexpected effects. Casting the model *before* saving it ensures that the tracer has the correct device information.

Q: How do I store attributes on a :class:`ScriptModule`?

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 583); *backlink*

Unknown interpreted text role "class".

Say we have a model like:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 587)

Unknown directive type "testcode".

```
.. testcode::

    import torch

    class Model(torch.nn.Module):
        def __init__(self):
            super(Model, self).__init__()
            self.x = 2

        def forward(self):
            return self.x

    m = torch.jit.script(Model())
```

If `Model` is instantiated it will result in a compilation error since the compiler doesn't know about `x`. There are 4 ways to inform the compiler of attributes on :class:`ScriptModule`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 603); *backlink*

Unknown interpreted text role "class".

1. `nn.Parameter` - Values wrapped in `nn.Parameter` will work as they do on `nn.Module`s

2. `register_buffer` - Values wrapped in `register_buffer` will work as they do on `nn.Module`s. This is equivalent to an attribute (see 4) of type `Tensor`.

3. Constants - Annotating a class member as `Final` (or adding it to a list called `__constants__` at the class definition level) will mark the contained names as constants. Constants are saved directly in the code of the model. See *builtin-constants* for details.

4. Attributes - Values that are a *supported type* can be added as mutable attributes. Most types can be inferred but some may need to be specified, see *module attributes* for details.

Q: I would like to trace module's method but I keep getting this error:

`RuntimeError: Cannot insert a Tensor that requires grad as a constant. Consider making it a parameter or input, or detaching the gradient`

This error usually means that the method you are tracing uses a module's parameters and you are passing the module's method instead of the module instance (e.g. `my_module_instance.forward` vs `my_module_instance`).

- Invoking `trace` with a module's method captures module parameters (which may require gradients) as **constants**.
- On the other hand, invoking `trace` with module's instance (e.g. `my_module`) creates a new module and correctly copies parameters into the new module, so they can accumulate gradients if required.

To trace a specific method on a module, see :func:`torch.jit.trace_module <torch.jit.trace_module>`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 633); *backlink*

Unknown interpreted text role "func".

Known Issues

If you're using `Sequential` with TorchScript, the inputs of some of the `Sequential` submodules may be falsely inferred to be `Tensor`, even if they're annotated otherwise. The canonical solution is to subclass `nn.Sequential` and redeclare `forward` with the input typed correctly.

Appendix

Migrating to PyTorch 1.2 Recursive Scripting API

This section details the changes to TorchScript in PyTorch 1.2. If you are new to TorchScript you can skip this section. There are two main changes to the TorchScript API with PyTorch 1.2.

1. :func:`torch.jit.script <torch.jit.script>` will now attempt to recursively compile functions, methods, and classes that it encounters. Once you call `torch.jit.script`, compilation is "opt-out", rather than "opt-in".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst`, **line 652**); *backlink*
>
> Unknown interpreted text role "func".

2. `torch.jit.script(nn_module_instance)` is now the preferred way to create :class:`ScriptModule`s, instead of inheriting from `torch.jit.ScriptModule`. These changes combine to provide a simpler, easier-to-use API for converting your `nn.Module`s into :class:`ScriptModule`s, ready to be optimized and executed in a non-Python environment.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst`, **line 656**); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst`, **line 656**); *backlink*
>
> Unknown interpreted text role "class".

The new usage looks like this:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst`, **line 664**)
>
> Unknown directive type "testcode".

```
.. testcode::

    import torch
    import torch.nn as nn
    import torch.nn.functional as F

    class Model(nn.Module):
        def __init__(self):
            super(Model, self).__init__()
            self.conv1 = nn.Conv2d(1, 20, 5)
            self.conv2 = nn.Conv2d(20, 20, 5)

        def forward(self, x):
            x = F.relu(self.conv1(x))
            return F.relu(self.conv2(x))

    my_model = Model()
    my_scripted_model = torch.jit.script(my_model)
```

- The module's `forward` is compiled by default. Methods called from `forward` are lazily compiled in the order they are used in `forward`.
- To compile a method other than `forward` that is not called from `forward`, add `@torch.jit.export`.
- To stop the compiler from compiling a method, add :func:`@torch.jit.ignore <torch.jit.ignore>` or :func:`@torch.jit.unused <torch.jit.unused>`. `@ignore` leaves the

  > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst`, **line 686**); *backlink*
  >
  > Unknown interpreted text role "func".

  > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst`, **line 686**); *backlink*
  >
  > Unknown interpreted text role "func".

- method as a call to python, and `@unused` replaces it with an exception. `@ignored` cannot be exported; `@unused` can.
- Most attribute types can be inferred, so `torch.jit.Attribute` is not necessary. For empty container types, annotate their types using PEP 526-style class annotations.
- Constants can be marked with a `Final` class annotation instead of adding the name of the member to `__constants__`.
- Python 3 type hints can be used in place of `torch.jit.annotate`

As a result of these changes, the following items are considered deprecated and should not appear in new code:

- The `@torch.jit.script_method` decorator
- Classes that inherit from `torch.jit.ScriptModule`
- The `torch.jit.Attribute` wrapper class
- The `__constants__` array
- The `torch.jit.annotate` function

Modules

> **Warning**
>
> The :func:`@torch.jit.ignore <torch.jit.ignore>` annotation's behavior changes in PyTorch 1.2. Before PyTorch 1.2 the @ignore decorator was used to make a function or method callable from code that is exported. To get this functionality back, use `@torch.jit.unused()`. `@torch.jit.ignore` is now equivalent to `@torch.jit.ignore(drop=False)`. See :func:`@torch.jit.ignore <torch.jit.ignore>` and :func:`@torch.jit.unused<torch.jit.unused>` for details.
>
> > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst`, **line 703**); *backlink*
> >
> > Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 703); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 703); *backlink*

Unknown interpreted text role "func".

When passed to the :func:`torch.jit.script <torch.jit.script>` function, a `torch.nn.Module`'s data is copied to a :class:`ScriptModule` and the TorchScript compiler compiles the module. The module's `forward` is compiled by default. Methods called from `forward` are lazily compiled in the order they are used in `forward`, as well as any `@torch.jit.export` methods.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 710); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 710); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 716)

Unknown directive type "autofunction".

```
.. autofunction:: export
```

Functions

Functions don't change much, they can be decorated with :func:`@torch.jit.ignore <torch.jit.ignore>` or :func:`torch.jit.unused <torch.jit.unused>` if needed.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 720); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 720); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 722)

Unknown directive type "testcode".

```
.. testcode::

    # Same behavior as pre-PyTorch 1.2
    @torch.jit.script
    def some_fn():
        return 2

    # Marks a function as ignored, if nothing
    # ever calls it then this has no effect
    @torch.jit.ignore
    def some_fn2():
        return 2

    # As with ignore, if nothing calls it then it has no effect.
    # If it is called in script it is replaced with an exception.
    @torch.jit.unused
    def some_fn3():
      import pdb; pdb.set_trace()
      return 4

    # Doesn't do anything, this function is already
    # the main entry point
    @torch.jit.export
    def some_fn4():
        return 2
```

TorchScript Classes

> **Warning**
>
> TorchScript class support is experimental. Currently it is best suited for simple record-like types (think a `NamedTuple` with methods attached).

Everything in a user defined TorchScript Class is exported by default, functions can be decorated with :func:`@torch.jit.ignore <torch.jit.ignore>` if needed.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 757); *backlink*

Unknown interpreted text role "func".

Attributes

The TorchScript compiler needs to know the types of *module attributes*. Most types can be inferred from the value of the member. Empty lists and dicts cannot have their types inferred and must have their types annotated with PEP 526-style class annotations. If a type cannot be inferred and is not explicitly annotated, it will not be added as an attribute to the resulting :class:`ScriptModule`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 763); *backlink*

Unknown interpreted text role "class".

Old API:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-

master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 772)

Unknown directive type "testcode".

```
.. testcode::

    from typing import Dict
    import torch

    class MyModule(torch.jit.ScriptModule):
        def __init__(self):
            super(MyModule, self).__init__()
            self.my_dict = torch.jit.Attribute({}, Dict[str, int])
            self.my_int = torch.jit.Attribute(20, int)

    m = MyModule()
```

New API:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 787)

Unknown directive type "testcode".

```
.. testcode::

    from typing import Dict

    class MyModule(torch.nn.Module):
        my_dict: Dict[str, int]

        def __init__(self):
            super(MyModule, self).__init__()
            # This type cannot be inferred and must be specified
            self.my_dict = {}

            # The attribute type here is inferred to be `int`
            self.my_int = 20

        def forward(self):
            pass

    m = torch.jit.script(MyModule())
```

Constants

The `Final` type constructor can be used to mark members as *constant*. If members are not marked constant, they will be copied to the resulting :class:`ScriptModule` as an attribute. Using `Final` opens opportunities for optimization if the value is known to be fixed and gives additional type safety.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 810); *backlink*

Unknown interpreted text role "class".

Old API:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 814)

Unknown directive type "testcode".

```
.. testcode::

    class MyModule(torch.jit.ScriptModule):
        __constants__ = ['my_constant']

        def __init__(self):
            super(MyModule, self).__init__()
            self.my_constant = 2

        def forward(self):
            pass
    m = MyModule()
```

New API:

```
try:
    from typing_extensions import Final
except:
    # If you don't have `typing_extensions` installed, you can use a
    # polyfill from `torch.jit`.
    from torch.jit import Final

class MyModule(torch.nn.Module):

    my_constant: Final[int]

    def __init__(self):
        super(MyModule, self).__init__()
        self.my_constant = 2

    def forward(self):
        pass

m = torch.jit.script(MyModule())
```

Variables

Containers are assumed to have type `Tensor` and be non-optional (see *Default Types* for more information). Previously, `torch.jit.annotate` was used to tell the TorchScript compiler what the type should be. Python 3 style type hints are now supported.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 860)

Unknown directive type "testcode".

```
.. testcode::

    import torch
    from typing import Dict, Optional

    @torch.jit.script
    def make_dict(flag: bool):
        x: Dict[str, int] = {}
        x['hi'] = 2
        b: Optional[int] = None
        if flag:
            b = 2
```

```
        return x, b
```

References

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 876)

Unknown directive type "toctree".

```
.. toctree::
    :maxdepth: 1

    jit_python_reference
    jit_unsupported
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)jit.rst, line 884)

Unknown directive type "py:module".

```
.. py:module:: torch.jit.mobile
```