torch.onnx

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 6)

Unknown directive type "automodule".

```
.. automodule:: torch.onnx
```

Open Neural Network eXchange (ONNX) is an open standard format for representing machine learning models. The torch.onnx module can export PyTorch models to ONNX. The model can then be consumed by any of the many runtimes that support ONNX.

Example: AlexNet from PyTorch to ONNX

Here is a simple script which exports a pretrained AlexNet to an ONNX file named `alexnet.onnx`. The call to `torch.onnx.export` runs the model once to trace its execution and then exports the traced model to the specified file:

```
import torch
import torchvision

dummy_input = torch.randn(10, 3, 224, 224, device="cuda")
model = torchvision.models.alexnet(pretrained=True).cuda()

# Providing input and output names sets the display names for values
# within the model's graph. Setting these does not change the semantics
# of the graph; it is only for readability.
#
# The inputs to the network consist of the flat list of inputs (i.e.
# the values you would pass to the forward() method) followed by the
# flat list of parameters. You can partially specify names, i.e. provide
# a list here shorter than the number of inputs to the model, and we will
# only set that subset of names, starting from the beginning.
input_names = [ "actual_input_1" ] + [ "learned_%d" % i for i in range(16) ]
output_names = [ "output1" ]

torch.onnx.export(model, dummy_input, "alexnet.onnx", verbose=True, input_names=input_names, output_names=output_names)
```

The resulting `alexnet.onnx` file contains a binary protocol buffer which contains both the network structure and parameters of the model you exported (in this case, AlexNet). The argument `verbose=True` causes the exporter to print out a human-readable representation of the model:

```
# These are the inputs and parameters to the network, which have taken on
# the names we specified earlier.
graph(%actual_input_1 : Float(10, 3, 224, 224)
      %learned_0 : Float(64, 3, 11, 11)
      %learned_1 : Float(64)
      %learned_2 : Float(192, 64, 5, 5)
      %learned_3 : Float(192)
      # ---- omitted for brevity ----
      %learned_14 : Float(1000, 4096)
      %learned_15 : Float(1000)) {
  # Every statement consists of some output tensors (and their types),
  # the operator to be run (with its attributes, e.g., kernels, strides,
  # etc.), its input tensors (%actual_input_1, %learned_0, %learned_1)
  %17 : Float(10, 64, 55, 55) = onnx::Conv[dilations=[1, 1], group=1, kernel_shape=[11, 11], pads=[2, 2, 2, 2], strides=[4, 4]](%actual_
  %18 : Float(10, 64, 55, 55) = onnx::Relu(%17), scope: AlexNet/Sequential[features]/ReLU[1]
  %19 : Float(10, 64, 27, 27) = onnx::MaxPool[kernel_shape=[3, 3], pads=[0, 0, 0, 0], strides=[2, 2]](%18), scope: AlexNet/Sequential[fea
  # ---- omitted for brevity ----
  %29 : Float(10, 256, 6, 6) = onnx::MaxPool[kernel_shape=[3, 3], pads=[0, 0, 0, 0], strides=[2, 2]](%28), scope: AlexNet/Sequential[feat
  # Dynamic means that the shape is not known. This may be because of a
  # limitation of our implementation (which we would like to fix in a
  # future release) or shapes which are truly dynamic.
  %30 : Dynamic = onnx::Shape(%29), scope: AlexNet
  %31 : Dynamic = onnx::Slice[axes=[0], ends=[1], starts=[0]](%30), scope: AlexNet
  %32 : Long() = onnx::Squeeze[axes=[0]](%31), scope: AlexNet
  %33 : Long() = onnx::Constant[value={9216}](), scope: AlexNet
  # ---- omitted for brevity ----
  %output1 : Float(10, 1000) = onnx::Gemm[alpha=1, beta=1, broadcast=1, transB=1](%45, %learned_14, %learned_15), scope: AlexNet/Sequent
  return (%output1);
}
```

You can also verify the output using the ONNX library, which you can install using conda:

```
conda install -c conda-forge onnx
```

Then, you can run:

```
import onnx

# Load the ONNX model
model = onnx.load("alexnet.onnx")

# Check that the model is well formed
onnx.checker.check_model(model)

# Print a human readable representation of the graph
print(onnx.helper.printable_graph(model.graph))
```

You can also run the exported model with one of the many runtimes that support ONNX. For example after installing ONNX Runtime, you can load and run the model:

```
import onnxruntime as ort

ort_session = ort.InferenceSession("alexnet.onnx")

outputs = ort_session.run(
    None,
    {"actual_input_1": np.random.randn(10, 3, 224, 224).astype(np.float32)},
)
print(outputs[0])
```

Here is a more involved tutorial on exporting a model and running it with ONNX Runtime.

Tracing vs Scripting

Internally, `torch.onnx.export()` requires a :class:`torch.jit.ScriptModule` rather than a :class:`torch.nn.Module`. If the passed-in model is not already a `ScriptModule`, `export()` will use *tracing* to convert it to one:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 115); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 115); *backlink*

Unknown interpreted text role "class".

- **Tracing**: If `torch.onnx.export()` is called with a Module that is not already a `ScriptModule`, it first does the equivalent of :func:`torch.jit.trace`, which executes the model once with the given `args` and records all operations that happen during that execution. This means that if your model is dynamic, e.g., changes behavior depending on input data, the exported model will *not* capture this dynamic behavior. Similarly, a trace is likely to be valid only for a specific input size. We recommend examining the exported model and making sure the operators look reasonable. Tracing will unroll loops and if statements, exporting a static graph that is exactly the same as the traced run. If you want to export your model with dynamic control flow, you will need to use *scripting*.

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 119); *backlink*

  Unknown interpreted text role "func".

- **Scripting**: Compiling a model via scripting preserves dynamic control flow and is valid for inputs of different sizes. To use scripting:
  - Use :func:`torch.jit.script` to produce a `ScriptModule`.

    System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 132); *backlink*

    Unknown interpreted text role "func".

  - Call `torch.onnx.export()` with the `ScriptModule` as the model, and set the `example_outputs` arg. This is required so that the types and shapes of the outputs can be captured without executing the model.

See Introduction to TorchScript and TorchScript for more details, including how to compose tracing and scripting to suit the particular requirements of different models.

Avoiding Pitfalls

Avoid NumPy and built-in Python types

PyTorch models can be written using NumPy or Python types and functions, but during :ref:`tracing<tracing-vs-scripting>`, any variables of NumPy or Python types (rather than torch.Tensor) are converted to constants, which will produce the wrong result if those values should change depending on the inputs.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 148); *backlink*

Unknown interpreted text role "ref".

For example, rather than using numpy functions on numpy.ndarrays:

```
# Bad! Will be replaced with constants during tracing.
x, y = np.random.rand(1, 2), np.random.rand(1, 2)
np.concatenate((x, y), axis=1)
```

Use torch operators on torch.Tensors:

```
# Good! Tensor operations will be captured during tracing.
x, y = torch.randn(1, 2), torch.randn(1, 2)
torch.cat((x, y), dim=1)
```

And rather than using :func:`torch.Tensor.item` (which converts a Tensor to a Python built-in number):

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 166); *backlink*

Unknown interpreted text role "func".

```
# Bad! y.item() will be replaced with a constant during tracing.
def forward(self, x, y):
    return x.reshape(y.item(), -1)
```

Use torch's support for implicit casting of single-element tensors:

```
# Good! y will be preserved as a variable during tracing.
def forward(self, x, y):
    return x.reshape(y, -1)
```

Avoid Tensor.data

Using the Tensor.data field can produce an incorrect trace and therefore an incorrect ONNX graph. Use :func:`torch.Tensor.detach` instead. (Work is ongoing to remove Tensor.data entirely).

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 182); *backlink*

Unknown interpreted text role "func".

Avoid in-place operations when using tensor.shape in tracing mode

In tracing mode, shape values obtained from tensor.shape are traced as tensors, and share the same memory. This might cause a mismatch in values of the final outputs. As a workaround, avoid use of inplace operations in these scenarios. For example, in the model:

```
class Model(torch.nn.Module):
  def forward(self, states):
      batch_size, seq_length = states.shape[:2]
      real_seq_length = seq_length
      real_seq_length += 2
      return real_seq_length + seq_length
```

`real_seq_length` and `seq_length` share the same memory in tracing mode. This could be avoided by rewriting the inplace operation:

```
real_seq_length = real_seq_length + 2
```

Limitations

Types

- Only torch.Tensors, numeric types that can be trivially converted to torch.Tensors (e.g. float, int), and tuples and lists of those types are supported as model inputs or outputs. Dict and str inputs and outputs are accepted in :ref:`tracing<tracing-vs-scripting>` mode, but:

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 212); *backlink*

  Unknown interpreted text role "ref".

  - Any computation that depends on the value of a dict or a str input will be replaced with the constant value seen during the one traced execution.
  - Any output that is a dict will be silently replaced with a flattened sequence of its values (keys will be removed). E.g. `{"foo": 1, "bar": 2}` becomes `(1, 2)`.
  - Any output that is a str will be silently removed.
- Certain operations involving tuples and lists are not supported in :ref:`scripting<tracing-vs-scripting>` mode due to limited support in ONNX for nested sequences. In particular appending a tuple to a list is not supported. In tracing mode, the nested sequences will be flattened automatically during the tracing.

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 222); *backlink*

  Unknown interpreted text role "ref".

Differences in Operator Implementations

Due to differences in implementations of operators, running the exported model on different runtimes may produce different results from each other or from PyTorch. Normally these differences are numerically small, so this should only be a concern if your application is sensitive to these small differences.

Unsupported Tensor Indexing Patterns

Tensor indexing patterns that cannot be exported are listed below. If you are experiencing issues exporting a model that does not include any of the unsupported patterns below, please double check that you are exporting with the latest `opset_version`.

Reads / Gets

When indexing into a tensor for reading, the following patterns are not supported:

```
# Tensor indices that includes negative values.
data[torch.tensor([[1, 2], [2, -3]]), torch.tensor([-2, 3])]
# Workarounds: use positive index values.
```

Writes / Sets

When indexing into a Tensor for writing, the following patterns are not supported:

```
# Multiple tensor indices if any has rank >= 2
data[torch.tensor([[1, 2], [2, 3]]), torch.tensor([2, 3])] = new_data
# Workarounds: use single tensor index with rank >= 2,
#              or multiple consecutive tensor indices with rank == 1.

# Multiple tensor indices that are not consecutive
data[torch.tensor([2, 3]), :, torch.tensor([1, 2])] = new_data
# Workarounds: transpose `data` such that tensor indices are consecutive.

# Tensor indices that includes negative values.
data[torch.tensor([1, -2]), torch.tensor([-2, 3])] = new_data
# Workarounds: use positive index values.

# Implicit broadcasting required for new_data.
data[torch.tensor([[0, 2], [1, 1]]), 1:3] = new_data
# Workarounds: expand new_data explicitly.
# Example:
#   data shape: [3, 4, 5]
#   new_data shape: [5]
#   expected new_data shape after broadcasting: [2, 2, 2, 5]
```

Adding support for operators

When exporting a model that includes unsupported operators, you'll see an error message like:

```
RuntimeError: ONNX export failed: Couldn't export operator foo
```

When that happens, you'll need to either change the model to not use that operator, or add support for the operator.

Adding support for operators requires contributing a change to PyTorch's source code. See CONTRIBUTING for general instructions on that, and below for specific instructions on the code changes required for supporting an operator.

During export, each node in the TorchScript graph is visited in topological order. Upon visiting a node, the exporter tries to find a registered symbolic functions for that node. Symbolic functions are implemented in Python. A symbolic function for an op named `foo` would look something like:

```
def foo(
  g: torch._C.Graph,
  input_0: torch._C.Value,
  input_1: torch._C.Value) -> Union[None, torch._C.Value, List[torch._C.Value]]:
  """
  Modifies g (e.g., using "g.op()"), adding the ONNX operations representing
  this PyTorch function.

  Args:
    g (Graph): graph to write the ONNX representation into.
    input_0 (Value): value representing the variables which contain
        the first input for this operator.
    input_1 (Value): value representing the variables which contain
        the second input for this operator.

  Returns:
    A Value or List of Values specifying the ONNX nodes that compute something
    equivalent to the original PyTorch operator with the given inputs.
    Returns None if it cannot be converted to ONNX.
  """
  ...
```

The `torch._C` types are Python wrappers around the types defined in C++ in ir.h.

The process for adding a symbolic function depends on the type of operator.

ATen operators

ATen is PyTorchâ€™s built-in tensor library. If the operator is an ATen operator (shows up in the TorchScript graph with the prefix `aten::`), make sure it is not supported already.

List of supported operators

Visit the auto generated :doc:`list of supported ATen operators <../onnx_supported_aten_ops>` for details on which operator are supported in each `opset_version`.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, **line 342); *backlink***
>
> Unknown interpreted text role "doc".

Adding support for an operator

If the operator is not in the list above:

- Define the symbolic function in `torch/onnx/symbolic_opset<version>.py`, for example torch/onnx/symbolic_opset9.py. Make sure the function has the same name as the ATen function, which may be declared in `torch/_C/_VariableFunctions.pyi` or `torch/nn/functional.pyi` (these files are generated at build time, so will not appear in your checkout until you build PyTorch).
- By default, the first arg is the ONNX graph. Other arg names must EXACTLY match the names in the `.pyi` file, because dispatch is done with keyword arguments.
- A symbolic function that has a first arg (before the Graph object) with the type annotation of torch.onnx.SymbolicContext will be called with that additional context. See examples below.
- In the symbolic function, if the operator is in the ONNX standard operator set, we only need to create a node to represent the ONNX operator in the graph. If not, we can create a graph of several standard operators that have equivalent semantics to the ATen operator.
- If an input argument is a Tensor, but ONNX asks for a scalar, we have to explicitly do the conversion. :func:`symbolic_helper._scalar` can convert a scalar tensor into a python scalar, and :func:`symbolic_helper._if_scalar_type_as` can turn a Python scalar into a PyTorch tensor.

  > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, **line 366); *backlink***
  >
  > Unknown interpreted text role "func".

  > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, **line 366); *backlink***
  >
  > Unknown interpreted text role "func".

Here is an example of handling missing symbolic function for the `ELU` operator.

If we run the following code:

```
print(
  torch.jit.trace(torch.nn.ELU(), # module
                  torch.ones(1)   # example input
                  ).graph)
```

We see something like:

```
graph(%self : __torch__.torch.nn.modules.activation.___torch_mangle_0.ELU,
      %input : Float(1, strides=[1], requires_grad=0, device=cpu)):
  %4 : float = prim::Constant[value=1.]()
  %5 : int = prim::Constant[value=1]()
  %6 : int = prim::Constant[value=1]()
  %7 : Float(1, strides=[1], requires_grad=0, device=cpu) = aten::elu(%input, %4, %5, %6)
  return (%7)
```

Since we see `aten::elu` in the graph, we know this is an ATen operator.

We check the ONNX operator list, and confirm that `Elu` is standardized in ONNX.

We find a signature for `elu` in `torch/nn/functional.pyi`:

```
def elu(input: Tensor, alpha: float = ..., inplace: bool = ...) -> Tensor: ...
```

We add the following lines to `symbolic_opset9.py`:

```
def elu(g, input, alpha, inplace=False):
    return g.op("Elu", input, alpha_f=_scalar(alpha))
```

Now PyTorch is able to export models containing the `aten::elu` operator!

See the `symbolic_opset*.py` files for more examples.

torch.autograd.Functions

If the operator is a sub-class of :class:`torch.autograd.Function`, there are two ways to export it.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, **line 413); *backlink***
>
> Unknown interpreted text role "class".

Static Symbolic Method

You can add a static method named `symbolic` to your function class. It should return ONNX operators that represent the function's behavior in ONNX. For example:

```
class MyRelu(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input: torch.Tensor) -> torch.Tensor:
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def symbolic(g: torch._C.graph, input: torch._C.Value) -> torch._C.Value:
        return g.op("Clip", input, g.op("Constant", value_t=torch.tensor(0, dtype=torch.float)))
```

PythonOp Symbolic

Alternatively, you can register a custom symbolic function. This gives the symbolic function access to more info through the `torch.onnx.SymbolicContext` object, which gets passed in as the first argument (before the `Graph` object).

All autograd `Function`s appear in the TorchScript graph as `prim::PythonOp` nodes. In order to differentiate between different `Function` subclasses, the symbolic function should use the `name` kwarg which gets set to the name of the class.

Custom symbolic functions should add type and shape information by calling `setType(...)` on Value objects before returning them (implemented in C++ by `torch::jit::Value::setType`). This is not required, but it can help the exporter's shape and type inference for down-stream nodes. For a non-trivial example of `setType`, see `test_aten_embedding_2` in test_operators.py.

The example below shows how you can access `requires_grad` via the `Node` object:

```
class MyClip(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, min):
        ctx.save_for_backward(input)
        return input.clamp(min=min)

class MyRelu(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        ctx.save_for_backward(input)
        return input.clamp(min=0)

def symbolic_python_op(ctx: torch.onnx.SymbolicContext, g: torch._C.Graph, *args, **kwargs):
    n = ctx.cur_node
    print("original node: ", n)
    for i, out in enumerate(n.outputs()):
        print("original output {}: {}, requires grad: {}".format(i, out, out.requiresGrad()))
    import torch.onnx.symbolic_helper as sym_helper
    for i, arg in enumerate(args):
        requires_grad = arg.requiresGrad() if sym_helper._is_value(arg) else False
        print("arg {}: {}, requires grad: {}".format(i, arg, requires_grad))

    name = kwargs["name"]
    ret = None
    if name == "MyClip":
        ret = g.op("Clip", args[0], args[1])
    elif name == "MyRelu":
        ret = g.op("Relu", args[0])
    else:
        # Logs a warning and returns None
        return _unimplemented("prim::PythonOp", "unknown node kind: " + name)
    # Copy type and shape from original node.
    ret.setType(n.type())
    return ret

from torch.onnx import register_custom_op_symbolic
register_custom_op_symbolic("prim::PythonOp", symbolic_python_op, 1)
```

Custom operators

If a model uses a custom operator implemented in C++ as described in Extending TorchScript with Custom C++ Operators, you can export it by following this example:

```
from torch.onnx import register_custom_op_symbolic
from torch.onnx.symbolic_helper import parse_args

# Define custom symbolic function
@parse_args("v", "v", "f", "i")
def symbolic_foo_forward(g, input1, input2, attr1, attr2):
    return g.op("custom_domain::Foo", input1, input2, attr1_f=attr1, attr2_i=attr2)

# Register custom symbolic function
register_custom_op_symbolic("custom_ops::foo_forward", symbolic_foo_forward, 9)

class FooModel(torch.nn.Module):
    def __init__(self, attr1, attr2):
        super(FooModule, self).__init__()
        self.attr1 = attr1
        self.attr2 = attr2

    def forward(self, input1, input2):
        # Calling custom op
        return torch.ops.custom_ops.foo_forward(input1, input2, self.attr1, self.attr2)

model = FooModel(attr1, attr2)
torch.onnx.export(
  model,
  (example_input1, example_input1),
  "model.onnx",
  # only needed if you want to specify an opset version > 1.
  custom_opsets={"custom_domain": 2})
```

You can export it as one or a combination of standard ONNX ops, or as a custom operator. The example above exports it as a custom operator in the "custom_domain" opset. When exporting a custom operator, you can specify the custom domain version using the `custom_opsets` dictionary at export. If not specified, the custom opset version defaults to 1. The runtime that consumes the model needs to support the custom op. See Caffe2 custom ops, ONNX Runtime custom ops, or your runtime of choice's documentation.

Discovering all unconvertible ATen ops at once

When export fails due to an unconvertible ATen op, there may in fact be more than one such op but the error message only mentions the first. To discover all of the unconvertible ops in one go you can:

```
from torch.onnx import utils as onnx_utils

# prepare model, args, opset_version
...

torch_script_graph, unconvertible_ops = onnx_utils.unconvertible_ops(
    model, args, opset_version=opset_version)

print(set(unconvertible_ops))
```

Frequently Asked Questions

Q: I have exported my LSTM model, but its input size seems to be fixed?

> The tracer records the shapes of the example inputs. If the model should accept inputs of dynamic shapes, set `dynamic_axes` when calling :func:`torch.onnx.export`.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 558); *backlink***
>
> Unknown interpreted text role "func".

Q: How to export models containing loops?

> See Tracing vs Scripting.

Q: How to export models with primitive type inputs (e.g. int, float)?

Support for primitive numeric type inputs was added in PyTorch 1.9. However, the exporter does not support models with str inputs.

Q: Does ONNX support implicit scalar datatype casting?

No, but the exporter will try to handle that part. Scalars are exported as constant tensors. The exporter will try to figure out the right datatype for scalars. However when it is unable to do so, you will need to manually specify the datatype. This often happens with scripted models, where the datatypes are not recorded. For example:

```
class ImplicitCastType(torch.jit.ScriptModule):
    @torch.jit.script_method
    def forward(self, x):
        # Exporter knows x is float32, will export "2" as float32 as well.
        y = x + 2
        # Currently the exporter doesn't know the datatype of y, so
        # "3" is exported as int64, which is wrong!
        return y + 3
        # To fix, replace the line above with:
        # return y + torch.tensor([3], dtype=torch.float32)

x = torch.tensor([1.0], dtype=torch.float32)
torch.onnx.export(ImplicitCastType(), x, "implicit_cast.onnx",
                  example_outputs=ImplicitCastType()(x))
```

We are trying to improve the datatype propagation in the exporter such that implicit casting is supported in more cases.

Q: Are lists of Tensors exportable to ONNX?

Yes, for `opset_version` >= 11, since ONNX introduced the Sequence type in opset 11.

Contributing / developing

Developer docs.

Functions

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 606)

Unknown directive type "autofunction".

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

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 607)

Unknown directive type "autofunction".

```
.. autofunction:: export_to_pretty_string
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 608)

Unknown directive type "autofunction".

```
.. autofunction:: register_custom_op_symbolic
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 609)

Unknown directive type "autofunction".

```
.. autofunction:: select_model_mode_for_export
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 610)

Unknown directive type "autofunction".

```
.. autofunction:: is_in_onnx_export
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 611)

Unknown directive type "autofunction".

```
.. autofunction:: is_onnx_log_enabled
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 612)

Unknown directive type "autofunction".

```
.. autofunction:: enable_log
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 613)

Unknown directive type "autofunction".

```
.. autofunction:: disable_log
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 614)

Unknown directive type "autofunction".

```
.. autofunction:: set_log_stream
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 615)

Unknown directive type "autofunction".

```
.. autofunction:: log
```

Classes

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]onnx.rst`, line 620)

Unknown directive type "autosummary".

```
.. autosummary::
    :toctree: generated
    :nosignatures:
    :template: classtemplate.rst

    SymbolicContext
```