Guidance for Operator Developer

PyTorch’s operators sometimes require changes for different reasons (e.g. from improving their usability to fixing bugs). These changes can be backward compatibility (BC) breaking, where older programs will no longer run as expected (or at all) on the latest version of PyTorch (an old program / new runtime problem), or forward compatibility (FC) breaking, where new programs will not run on older versions of PyTorch (a new program / old runtime problem). This guidance focuses on the requirements for maintaining backwards comatibility when making changes to an operator. In order to do this we introduce the concept of the *upgrader*: a method to adapt the new operator to mimic the old operator behavior. When a new runtime reads an old program containing the old operator definition, the upgrader will adapt the old operator definition to comply with the new operator implementation. As you would expect, an upgrader is only applied when an old operation definition is encountered (i.e. if there are no "old" operators in the program, no upgrader would be used). For more details on the reasoning behind this new requirement please refer to the PyTorch Operator Versioning RFC.

If the change to the operator is BC-breaking in either the schema or the semantics, you are responsible for writing an upgrader to prevent the change from becoming BC breaking.

You can determine if your change in the operator is BC breaking, if it fails `test/forward_backward_compatibility/check_forward_backward_compatibility.py` .

Some examples BC breaking changes

When making changes to the operators, the first thing to identify is if it's BC/FC breaking. Again, we only targetting for BC breaking changes on this guidance. Here are some examples to help understanding what a BC changes may look like:

Backward Compatibility Breakage:

- Return types are more generic than the older version
  - Old: `foo(Tensor self, int a) -> int`
  - New: `foo(Tensor self, int a) -> Scalar`
- Argument types are more specific than the older version
  - Old: `foo(Tensor self, Scalar a) -> int`
  - New: `foo(Tensor self, int a) -> int`
- Added new arguments don’t have associated default values
  - Old: `foo(Tensor self, int a) -> int`
  - New: `foo(Tensor self, int a, int b) -> int`
- Internal implementation change even when the schema remains the same
- Deprecating an operator

The steps to write upgrader:

1.Preparation

Build PyTorch from souce and prepare a test model before making changes to the operator, following the process below. A test model before making the operator changes is needed to test the upgrader. Otherwise, after the change to operator, the new runtime will no longer be able to produce a model with the historic operator and can't test it anymore.

```
1. Add a test module in `test/jit/fixtures_srcs/fixtures_src.py`. In `test/jit/fixtures_srcs/generate_models.py`,
```

```
class TestVersionedLinspaceV7(torch.nn.Module):
    def __init__(self):
        super(TestVersionedLinspaceV7, self).__init__()

    def forward(self, a: Union[int, float, complex], b: Union[int, float, complex]):
        c = torch.linspace(a, b, steps=5)
        d = torch.linspace(a, b)
        return c, d
```

```
    Please make sure the module uses the changed operator and follow the name schema `
TestVersioned{${OpnameOverloadedname}}V${kProducedFileFormatVersion}`. [`kProducedFileFormatVersion`]
(https://github.com/pytorch/pytorch/blob/master/caffe2/serialize/versions.h#L82) can be found in `versions.h`. The example
operator usage can be found on [PyTorch Docs](https://pytorch.org/docs/stable/index.html), like [linspace operator]
(https://pytorch.org/docs/stable/generated/torch.linspace.html)
 2. Register its corresponding changed operator in ALL_MODULES like following. Use an instance as the key and the changed
operator as the value. It will ensure the test model covers everything needed. It's important to check in a valid test
model before making the change to the runtime, as it will be really challenging to switch to the revision of the source
code and regenerate the test model after the change is merged.
```

```
# key: test module instance, value: changed operator name
ALL_MODULES = {
    TestVersionedLinspaceV7(): "aten::linspace",
}
```

```
    This module should include the changed operator. If the operator isn't covered in the model, the model export process
will fail.
```

```
 3. Export the model to `test/jit/fixtures` by running
```

```
python test/jit/fixtures_src/generate_models.py
```

```
 4. Commit the change and submit a pull request.
```

2. Make changes to the operator and write an upgrader.

```
1. Make the operator change.
2. Write an upgrader in `torch/csrc/jit/operator_upgraders/upgraders_entry.cpp` file inside a map `kUpgradersEntryMap`. The softly enforced naming format is `<operator_name>_<operator_overload>_<start>_<end>`. The start and end means the upgrader can be applied to the operator exported during when [the global operator version](https://github.com/pytorch/pytorch/blob/master/caffe2/serialize/versions.h#L82) within the range `[start, end]`. Let's take an operator `linspace` with the overloaded name `out` as an example. The first thing is to check if the upgrader exists in in [upgraders_entry.cpp](https://github.com/pytorch/pytorch/blob/master/torch/csrc/jit/operator_upgraders/upgraders_entry.cpp).
    1. If the upgrader doesn't exist in `upgraders_entry.cpp`, the upgrader name can be `linspace_out_0_{kProducedFileFormatVersion}`, where [`kProducedFileFormatVersion`](https://github.com/pytorch/pytorch/blob/master/caffe2/serialize/versions.h#L82) can be found in [versions.h](https://github.com/pytorch/pytorch/blob/master/caffe2/serialize/versions.h).
    2. If the upgrader exist in `upgraders_entry.cpp`, for example `linspace_out_0_7` (means `linspace.out` operator is changed when operator version is bumped from 7 to 8),
        1. If it's possible to write an upgrader valid for `linspace` before versioning bumping to 8, after versioning bumping to 8, write an upgrader `linspace_out_0_{kProducedFileFormatVersion}`
        2. If it's impossible to write an upgrader valid for `linspace` before versioning bumping to 8, check the date when the version is bumped to 8  at [`versions.h`](https://github.com/pytorch/pytorch/blob/master/caffe2/serialize/versions.h#L82). If it has been 180 days, write an upgrader `linspace_out_8_{kProducedFileFormatVersion}` for `linspace.out` after bumping to 8, and deprecate the old upgrader. If it hasn't been 180 days, wait until 180 days and do the same changes as above.

To write an upgrader, you would need to know how the new runtime with the new `linspace` operator can handle an old model with the old `linspace` operator. When `linspace` is bumped to 8, the change is to make `step` a required argument, instead of an optional argument. The old schema is:
```

```
linspace(start: Union[int, float, complex], end: Union[int, float, complex], steps: Optional[int], dtype: Optional[int], layout: Optional[int],
                  device: Optional[Device], pin_memory: Optional[bool]):
```

```
And the new schema is:
```

```
linspace(start: Union[int, float, complex], end: Union[int, float, complex], steps: int, dtype: Optional[int], layout: Optional[int],
                  device: Optional[Device], pin_memory: Optional[bool]):
```

```
An upgrader will only be applied to an old model and it won't be applied to a new model. The upgrader can be written with the following logic:
```

```
def linspace_0_7(start: Union[int, float, complex], end: Union[int, float, complex], steps: Optional[int], *, dtype: Optional[int], layout: Optional[int],
                  device: Optional[Device], pin_memory: Optional[bool]):
  if (steps is None):
    return torch.linspace(start=start, end=end, steps=100, dtype=dtype, layout=layout, device=device, pin_memory=pin_memory)
  return torch.linspace(start=start, end=end, steps=steps, dtype=dtype, layout=layout, device=device, pin_memory=pin_memory)
```

```
The actual upgrader needs to be written as [TorchScript](https://pytorch.org/docs/stable/jit.html), and the below example is the actual upgrader of the operator `linspace.out `and the operator ` linspace` exported at version from 0 to 7.
```

```
static std::unordered_map<std::string, std::string> kUpgradersEntryMap(
    {
      {"linspace_0_7", R"SCRIPT(
def linspace_0_7(start: Union[int, float, complex], end: Union[int, float, complex], steps: Optional[int], *, dtype: Optional[int], layout: Optional[int],
                  device: Optional[Device], pin_memory: Optional[bool]):
  if (steps is None):
    return torch.linspace(start=start, end=end, steps=100, dtype=dtype, layout=layout, device=device, pin_memory=pin_memory)
  return torch.linspace(start=start, end=end, steps=steps, dtype=dtype, layout=layout, device=device, pin_memory=pin_memory)
)SCRIPT"},
    }
```

```
With the upgrader, when a new runtime loads an old model, it will first check the operator version of the old model. If
it's older than the current runtime, it will replace the operator from the old model with the upgrader above.

3. Bump [`kMaxSupportedFileFormatVersion`](https://github.com/pytorch/pytorch/blob/master/caffe2/serialize/versions.h#L15)
the [`kProducedFileFormatVersion`](https://github.com/pytorch/pytorch/blob/master/caffe2/serialize/versions.h#L82) by 1 and
provide the reasons under [`versions.h`](https://github.com/pytorch/pytorch/blob/master/caffe2/serialize/versions.h#L73-
L81)
```

```

constexpr uint64_t kMaxSupportedFileFormatVersion = 0x9L;

...
// We describe new operator version bump reasons here:
// 1) [01/24/2022]
//     We bump the version number to 8 to update aten::linspace
//     and aten::linspace.out to error out when steps is not
//     provided. (see: https://github.com/pytorch/pytorch/issues/55951)
// 2) [01/30/2022]
//     Bump the version number to 9 to update aten::logspace and
//     and aten::logspace.out to error out when steps is not
//     provided. (see: https://github.com/pytorch/pytorch/issues/55951)
constexpr uint64_t kProducedFileFormatVersion = 0x9L;
```

```
4. In `torch/csrc/jit/operator_upgraders/version_map.cpp`, add changes like below. You will need to make sure that the
entry is **SORTED** by the bumped to version number.
```

```
{{${operator_name.overloaded_name},
  {{${bump_to_version},
    "${upgrader_name}",
    "${old operator schema}"}}},
```

```
For the example operator `linspace`, if there are two version bumps, one is bumped to 8 and one is bumped to 12, the sorted
result is:
```

```
{{"aten::linspace",
  {{12,
    "linspace_0_11",
    "aten::linspace(Scalar start, Scalar end, int? steps=None, *, ScalarType? dtype=None, Layout? layout=None, Device?
device=None, bool? pin_memory=None) -> Tensor"}}},
  {{8,
    "linspace_0_7",
    "aten::linspace(Scalar start, Scalar end, int? steps=None, *, ScalarType? dtype=None, Layout? layout=None, Device?
device=None, bool? pin_memory=None) -> Tensor"}}},
```

```
5. After [rebuilding PyTorch](https://github.com/pytorch/pytorch#from-source), run the following command to auto update the
file [`torch/csrc/jit/mobile/upgrader_mobile.cpp`]
(https://github.com/pytorch/pytorch/blob/8757e21c6a4fc00e83539aa7f9c28eb11eff53c1/torch/csrc/jit/mobile/upgrader_mobile.cpp).
 After rebuild PyTorch from source (`python setup.py`), run
```

```
python pytorch/tools/codegen/operator_versions/gen_mobile_upgraders.py
```

```
6. Add a test. With the model generated from step 1, you will need to add tests in
`test/test_save_load_for_op_versions.py`. Following is an example to write a test
```

```
      @settings(max_examples=10, deadline=200000)  # A total of 10 examples will be generated
      @given(
          sample_input=st.tuples(st.integers(min_value=5, max_value=199), st.floats(min_value=5.0, max_value=199.0))
      )  # Generate a pair (integer, float)
      @example((2, 3, 2.0, 3.0))  # Ensure this example will be covered
      def test_versioned_div_scalar(self, sample_input):
          # Step 1. Write down the old behavior of this operator, if possible
          def historic_div_scalar_float(self, other: float):
              return torch.true_divide(self, other)

          # Step 2. Write down how current module should look like
          class MyModuleFloat(torch.nn.Module):
              def __init__(self):
                  super(MyModuleFloat, self).__init__()
```

```
            def forward(self, a, b: float):
                return a / b
        try:
            # Step 3. Load the old model and it will apply upgrader
            v3_mobile_module_float = _load_for_lite_interpreter(
                pytorch_test_dir + "/jit/fixtures/test_versioned_div_scalar_float_v2.ptl")
            v3_server_module_float = torch.jit.load(
                pytorch_test_dir + "/jit/fixtures/test_versioned_div_scalar_float_v2.ptl")
        except Exception as e:
            self.skipTest("Failed to load fixture!")

        # Step4. Load the new model and it won't apply the ugprader
        current_mobile_module_float = self._save_load_mobile_module(MyModuleFloat)
        current_server_module_float = self._save_load_module(MyModuleFloat)

        for val_a, val_b in product(sample_input, sample_input):
            a = torch.tensor((val_a,))
            b = val_b

            def _helper(m, fn):
                m_result = self._try_fn(m, a, b)
                fn_result = self._try_fn(fn, a, b)

                if isinstance(m_result, Exception):
                    self.assertTrue(fn_result, Exception)
                else:
                    self.assertEqual(m_result, fn_result)

            # Ensure the module loaded from the old model with upgrader
            # has the same result as the module loaded from the new model
            _helper(v3_mobile_module_float, current_mobile_module_float)
            _helper(v3_mobile_module_float, current_server_module_float)

            # Ensure the module loaded from the new model with upgrader
            # has the same result as the module loaded from the new model
            _helper(current_mobile_module_float, torch.div)
            _helper(current_server_module_float, torch.div)
```

```
7. Commit all changes made in step 2 in a single pull request and submit it.
```

You can look at following PRs to get the rough idea of what needs to be done:

1. PR that adds `logspace` test modules
2. PR that updates `logspace`

NOTE

1. Adding arguments with a default value to an operator is not BC breaking, and thus does not require an upgrader. For example, the following change to operator `foo` is backwards compatible:

```
# before
def foo(x, y):
    return x, y
```

```
# after
def foo(x, y, z=100):
    return x, y, z
```

2. To help understanding the BC/FC breakage changes, here are some FC breaking changes examples. The solution to resolve it is not there yet. If it's desired, please report it in either PyTorch Forum or PyTorch Github. We will prioritize it accordingly.

   - Adding new default argument:
   - Adding a new default argument not RIGHT BEFORE the out arguments which can be 0 or more.
     - Old: `foo(Tensor self, int a, int b=1, Tensor(a!) out) -> (Tensor(a!))`
     - New: `foo(Tensor self, int a, int c=1, int b=1, Tensor(a!) out) -> (Tensor(a!))`
   - Adding out argument NOT at the end of the schema.
     - Old: `foo(Tensor self, int a, int b=1, Tensor(a!) out) -> (Tensor(a!))`
     - New: `foo(Tensor self, int a, Tensor(d!), int b=1, Tensor(a!) out) -> (Tensor(a!), Tensor(d!))`
   - Adding default arguments with container types such as ListType or DictType (list or dict).

        - Old: `foo(Tensor self, int a, int b=1, Tensor(a!) out) -> (Tensor(a!))`
        - New: `foo(Tensor self, int a, int b=1, int[2] c=1, Tensor(a!) out) -> (Tensor(a!))`
    - Changing default argument's name
        - This will only work when the default argument always uses the default value (so that serialization will ignore it). In all other cases, it will fail.
        - Old: `foo(Tensor self, int a, int b=1, Tensor(a!) out) -> (Tensor(a!))`
        - New: `foo(Tensor self, int a, int c=1, Tensor(a!) out) -> (Tensor(a!))`
    - Changing default argument's default value. This will break when this argument is saved with the default value in newer runtime. Older runtime will use its old default value which will lead to wrong output.
        - Old: `foo(Tensor self, int a, int b=1, Tensor(a!) out) -> (Tensor(a!))`
        - New: `foo(Tensor self, int a, int b=4, Tensor(a!) out) -> (Tensor(a!))`
    - Adding new operator
