Modules

PyTorch uses modules to represent neural networks. Modules are:

- **Building blocks of stateful computation.** PyTorch provides a robust library of modules and makes it simple to define new custom modules, allowing for easy construction of elaborate, multi-layer neural networks.
- **Tightly integrated with PyTorch's** autograd **system.** Modules make it simple to specify learnable parameters for PyTorch's Optimizers to update.
- **Easy to work with and transform.** Modules are straightforward to save and restore, transfer between CPU / GPU / TPU devices, prune, quantize, and more.

This note describes modules, and is intended for all PyTorch users. Since modules are so fundamental to PyTorch, many topics in this note are elaborated on in other notes or tutorials, and links to many of those documents are provided here as well.

A Simple Custom Module

To get started, let's look at a simpler, custom version of PyTorch's :class:`~torch.nn.Linear` module. This module applies an affine transformation to its input.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 26); *backlink***
>
> Unknown interpreted text role "class".

```
import torch
from torch import nn

class MyLinear(nn.Module):
  def __init__(self, in_features, out_features):
    super().__init__()
    self.weight = nn.Parameter(torch.randn(in_features, out_features))
    self.bias = nn.Parameter(torch.randn(out_features))

  def forward(self, input):
    return (input @ self.weight) + self.bias
```

This simple module has the following fundamental characteristics of modules:

- **It inherits from the base Module class.** All modules should subclass :class:`~torch.nn.Module` for composability with other modules.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 45); *backlink***
>
> Unknown interpreted text role "class".

- **It defines some "state" that is used in computation.** Here, the state consists of randomly-initialized `weight` and `bias` tensors that define the affine transformation. Because each of these is defined as a :class:`~torch.nn.parameter.Parameter`, they are *registered* for the module and will automatically be tracked and returned from calls to :func:`~torch.nn.Module.parameters`. Parameters can be considered the "learnable" aspects of the module's computation (more on this later). Note that modules are not required to have state, and can also be stateless.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 47); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 47); *backlink*

Unknown interpreted text role "func".

- **It defines a forward() function that performs the computation.** For this affine transformation module, the input is matrix-multiplied with the `weight` parameter (using the `@` short-hand notation) and added to the `bias` parameter to produce the output. More generally, the `forward()` implementation for a module can perform arbitrary computation involving any number of inputs and outputs.

This simple module demonstrates how modules package state and computation together. Instances of this module can be constructed and called:

```
m = MyLinear(4, 3)
sample_input = torch.randn(4)
m(sample_input)
: tensor([-0.3037, -1.0413, -4.2057], grad_fn=<AddBackward0>)
```

Note that the module itself is callable, and that calling it invokes its `forward()` function. This name is in reference to the concepts of "forward pass" and "backward pass", which apply to each module. The "forward pass" is responsible for applying the computation represented by the module to the given input(s) (as shown in the above snippet). The "backward pass" computes gradients of module outputs with respect to its inputs, which can be used for "training" parameters through gradient descent methods. PyTorch's autograd system automatically takes care of this backward pass computation, so it is not required to manually implement a `backward()` function for each module. The process of training module parameters through successive forward / backward passes is covered in detail in :ref`Neural Network Training with Modules`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 69); *backlink*

Unknown interpreted text role "ref".

The full set of parameters registered by the module can be iterated through via a call to :func:`~torch.nn.Module.parameters` or :func:`~torch.nn.Module.named_parameters`, where the latter includes each parameter's name:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 79); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 79); *backlink*

Unknown interpreted text role "func".

```
for parameter in m.named_parameters():
  print(parameter)
: ('weight', Parameter containing:
tensor([[ 1.0597,  1.1796,  0.8247],
        [-0.5080, -1.2635, -1.1045],
        [ 0.0593,  0.2469, -1.4299],
        [-0.4926, -0.5457,  0.4793]], requires_grad=True))
('bias', Parameter containing:
tensor([ 0.3634,  0.2015, -0.8525], requires_grad=True))
```

In general, the parameters registered by a module are aspects of the module's computation that should be "learned". A later section of this note shows how to update these parameters using one of PyTorch's Optimizers. Before we get to that, however, let's first examine how modules can be composed with one another.

Modules as Building Blocks

Modules can contain other modules, making them useful building blocks for developing more elaborate functionality. The simplest way to do this is using the :class:`~torch.nn.Sequential` module. It allows us to chain together multiple modules:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 102); *backlink*

Unknown interpreted text role "class".

```
net = nn.Sequential(
  MyLinear(4, 3),
  nn.ReLU(),
  MyLinear(3, 1)
)

sample_input = torch.randn(4)
net(sample_input)
: tensor([-0.6749], grad_fn=<AddBackward0>)
```

Note that :class:`~torch.nn.Sequential` automatically feeds the output of the first `MyLinear` module as input into the :class:`~torch.nn.ReLU`, and the output of that as input into the second `MyLinear` module. As shown, it is limited to in-order chaining of modules with a single input and output.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 118); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 118); *backlink*

Unknown interpreted text role "class".

In general, it is recommended to define a custom module for anything beyond the simplest use cases, as this gives full flexibility on how submodules are used for a module's computation.

For example, here's a simple neural network implemented as a custom module:

```
import torch.nn.functional as F

class Net(nn.Module):
  def __init__(self):
    super().__init__()
    self.l0 = MyLinear(4, 3)
    self.l1 = MyLinear(3, 1)
  def forward(self, x):
    x = self.l0(x)
    x = F.relu(x)
    x = self.l1(x)
    return x
```

This module is composed of two "children" or "submodules" (`l0` and `l1`) that define the layers of the neural network and are utilized for computation within the module's `forward()` method. Immediate children of a module can be iterated through via a call to :func:`~torch.nn.Module.children` or :func:`~torch.nn.Module.named_children`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 142); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 142); *backlink*

Unknown interpreted text role "func".

```
net = Net()
```

```
for child in net.named_children():
  print(child)
: ('l0', MyLinear())
('l1', MyLinear())
```

To go deeper than just the immediate children, :func:`~torch.nn.Module.modules` and :func:`~torch.nn.Module.named_modules` *recursively* iterate through a module and its child modules:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 155); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 155); *backlink*

Unknown interpreted text role "func".

```
class BigNet(nn.Module):
  def __init__(self):
    super().__init__()
    self.l1 = MyLinear(5, 4)
    self.net = Net()
  def forward(self, x):
    return self.net(self.l1(x))

big_net = BigNet()
for module in big_net.named_modules():
  print(module)
: ('', BigNet(
  (l1): MyLinear()
  (net): Net(
    (l0): MyLinear()
    (l1): MyLinear()
  )
))
('l1', MyLinear())
('net', Net(
  (l0): MyLinear()
  (l1): MyLinear()
))
('net.l0', MyLinear())
('net.l1', MyLinear())
```

Sometimes, it's necessary for a module to dynamically define submodules. The :class:`~torch.nn.ModuleList` and :class:`~torch.nn.ModuleDict` modules are useful here; they register submodules from a list or dict:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 186); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 186); *backlink*

Unknown interpreted text role "class".

```
class DynamicNet(nn.Module):
  def __init__(self, num_layers):
    super().__init__()
    self.linears = nn.ModuleList(
      [MyLinear(4, 4) for _ in range(num_layers)])
    self.activations = nn.ModuleDict({
      'relu': nn.ReLU(),
      'lrelu': nn.LeakyReLU()
    })
    self.final = MyLinear(4, 1)
  def forward(self, x, act):
    for linear in self.linears:
      x = linear(x)
```

```
    x = self.activations[act](x)
    x = self.final(x)
    return x

dynamic_net = DynamicNet(3)
sample_input = torch.randn(4)
output = dynamic_net(sample_input, 'relu')
```

For any given module, its parameters consist of its direct parameters as well as the parameters of all submodules. This means that calls to :func:`~torch.nn.Module.parameters` and :func:`~torch.nn.Module.named_parameters` will recursively include child parameters, allowing for convenient optimization of all parameters within the network:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 213); ***backlink***

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 213); ***backlink***

Unknown interpreted text role "func".

```
for parameter in dynamic_net.named_parameters():
  print(parameter)
: ('linears.0.weight', Parameter containing:
tensor([[-1.2051,  0.7601,  1.1065,  0.1963],
        [ 3.0592,  0.4354,  1.6598,  0.9828],
        [-0.4446,  0.4628,  0.8774,  1.6848],
        [-0.1222,  1.5458,  1.1729,  1.4647]], requires_grad=True))
('linears.0.bias', Parameter containing:
tensor([ 1.5310,  1.0609, -2.0940,  1.1266], requires_grad=True))
('linears.1.weight', Parameter containing:
tensor([[ 2.1113, -0.0623, -1.0806,  0.3508],
        [-0.0550,  1.5317,  1.1064, -0.5562],
        [-0.4028, -0.6942,  1.5793, -1.0140],
        [-0.0329,  0.1160, -1.7183, -1.0434]], requires_grad=True))
('linears.1.bias', Parameter containing:
tensor([ 0.0361, -0.9768, -0.3889,  1.1613], requires_grad=True))
('linears.2.weight', Parameter containing:
tensor([[-2.6340, -0.3887, -0.9979,  0.0767],
        [-0.3526,  0.8756, -1.5847, -0.6016],
        [-0.3269, -0.1608,  0.2897, -2.0829],
        [ 2.6338,  0.9239,  0.6943, -1.5034]], requires_grad=True))
('linears.2.bias', Parameter containing:
tensor([ 1.0268,  0.4489, -0.9403,  0.1571], requires_grad=True))
('final.weight', Parameter containing:
tensor([[ 0.2509], [-0.5052], [ 0.3088], [-1.4951]], requires_grad=True))
('final.bias', Parameter containing:
tensor([0.3381], requires_grad=True))
```

It's also easy to move all parameters to a different device or change their precision using :func:`~torch.nn.Module.to`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 247); ***backlink***

Unknown interpreted text role "func".

```
# Move all parameters to a CUDA device
dynamic_net.to(device='cuda')

# Change precision of all parameters
dynamic_net.to(dtype=torch.float64)

dynamic_net(torch.randn(5, device='cuda', dtype=torch.float64))
: tensor([6.5166], device='cuda:0', dtype=torch.float64, grad_fn=<AddBackward0>)
```

More generally, an arbitrary function can be applied to a module and its submodules recursively by using the :func:`~torch.nn.Module.apply` function. For example, to apply custom initialization to parameters of a module and its submodules:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 261);

backlink

Unknown interpreted text role "func".

```
# Define a function to initialize Linear weights.
# Note that no_grad() is used here to avoid tracking this computation in the autograd graph.
@torch.no_grad()
def init_weights(m):
  if isinstance(m, nn.Linear):
    nn.init.xavier_normal_(m.weight)
    m.bias.fill_(0.0)

# Apply the function recursively on the module and its submodules.
dynamic_net.apply(init_weights)
```

These examples show how elaborate neural networks can be formed through module composition and conveniently manipulated. To allow for quick and easy construction of neural networks with minimal boilerplate, PyTorch provides a large library of performant modules within the :mod:`torch.nn` namespace that perform common neural network operations like pooling, convolutions, loss functions, etc.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 278);
backlink

Unknown interpreted text role "mod".

In the next section, we give a full example of training a neural network.

For more information, check out:

- Library of PyTorch-provided modules: torch.nn
- Defining neural net modules: https://pytorch.org/tutorials/beginner/examples_nn/polynomial_module.html

Neural Network Training with Modules

Once a network is built, it has to be trained, and its parameters can be easily optimized with one of PyTorchâ€™s Optimizers from :mod:`torch.optim`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 295);
backlink

Unknown interpreted text role "mod".

```
# Create the network (from previous section) and optimizer
net = Net()
optimizer = torch.optim.SGD(net.parameters(), lr=1e-4, weight_decay=1e-2, momentum=0.9)

# Run a sample training loop that "teaches" the network
# to output the constant zero function
for _ in range(10000):
  input = torch.randn(4)
  output = net(input)
  loss = torch.abs(output)
  net.zero_grad()
  loss.backward()
  optimizer.step()

# After training, switch the module to eval mode to do inference, compute performance metrics, etc.
# (see discussion below for a description of training and evaluation modes)
...
net.eval()
...
```

In this simplified example, the network learns to simply output zero, as any non-zero output is "penalized" according to its absolute value by employing :func:`torch.abs` as a loss function. While this is not a very interesting task, the key parts of training are present:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 320);
backlink

Unknown interpreted text role "func".

- A network is created.
- An optimizer (in this case, a stochastic gradient descent optimizer) is created, and the networkâ€™s parameters are associated with it.
- A training loop...
    - acquires an input,
    - runs the network,
    - computes a loss,
    - zeros the networkâ€™s parametersâ€™ gradients,
    - calls loss.backward() to update the parametersâ€™ gradients,
    - calls optimizer.step() to apply the gradients to the parameters.

After the above snippet has been run, note that the network's parameters have changed. In particular, examining the value of `l1`'s `weight` parameter shows that its values are now much closer to 0 (as may be expected):

```
print(net.l1.weight)
: Parameter containing:
tensor([[-0.0013],
        [ 0.0030],
        [-0.0008]], requires_grad=True)
```

Note that the above process is done entirely while the network module is in "training mode". Modules default to training mode and can be switched between training and evaluation modes using :func:`~torch.nn.Module.train` and :func:`~torch.nn.Module.eval`. They can behave differently depending on which mode they are in. For example, the :class:`~torch.nn.BatchNorm` module maintains a running mean and variance during training that are not updated when the module is in evaluation mode. In general, modules should be in training mode during training and only switched to evaluation mode for inference or evaluation. Below is an example of a custom module that behaves differently between the two modes:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 346); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 346); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 346); *backlink*

Unknown interpreted text role "class".

```
class ModalModule(nn.Module):
  def __init__(self):
    super().__init__()

  def forward(self, x):
    if self.training:
      # Add a constant only in training mode.
      return x + 1.
    else:
      return x


m = ModalModule()
x = torch.randn(4)

print('training mode output: {}'.format(m(x)))
: tensor([1.6614, 1.2669, 1.0617, 1.6213, 0.5481])

m.eval()
print('evaluation mode output: {}'.format(m(x)))
: tensor([ 0.6614,  0.2669,  0.0617,  0.6213, -0.4519])
```

Training neural networks can often be tricky. For more information, check out:

- Using Optimizers: https://pytorch.org/tutorials/beginner/examples_nn/two_layer_net_optim.html.
- Neural network training: https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html
- Introduction to autograd: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

Module State

In the previous section, we demonstrated training a module's "parameters", or learnable aspects of computation. Now, if we want to save the trained model to disk, we can do so by saving its `state_dict` (i.e. "state dictionary"):

```
# Save the module
torch.save(net.state_dict(), 'net.pt')

...

# Load the module later on
new_net = Net()
new_net.load_state_dict(torch.load('net.pt'))
: <All keys matched successfully>
```

A module's `state_dict` contains state that affects its computation. This includes, but is not limited to, the module's parameters. For some modules, it may be useful to have state beyond parameters that affects module computation but is not learnable. For such cases, PyTorch provides the concept of "buffers", both "persistent" and "non-persistent". Following is an overview of the various types of state a module can have:

- **Parameters**: learnable aspects of computation; contained within the `state_dict`
- **Buffers**: non-learnable aspects of computation
    - **Persistent** buffers: contained within the `state_dict` (i.e. serialized when saving & loading)
    - **Non-persistent** buffers: not contained within the `state_dict` (i.e. left out of serialization)

As a motivating example for the use of buffers, consider a simple module that maintains a running mean. We want the current value of the running mean to be considered part of the module's `state_dict` so that it will be restored when loading a serialized form of the module, but we don't want it to be learnable. This snippet shows how to use :func:`~torch.nn.Module.register_buffer` to accomplish this:

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

```
class RunningMean(nn.Module):
  def __init__(self, num_features, momentum=0.9):
    super().__init__()
    self.momentum = momentum
    self.register_buffer('mean', torch.zeros(num_features))
  def forward(self, x):
    self.mean = self.momentum * self.mean + (1.0 - self.momentum) * x
    return self.mean
```

Now, the current value of the running mean is considered part of the module's `state_dict` and will be properly restored when loading the module from disk:

```
m = RunningMean(4)
for _ in range(10):
  input = torch.randn(4)
  m(input)

print(m.state_dict())
: OrderedDict([('mean', tensor([ 0.1041, -0.1113, -0.0647,  0.1515]))]))

# Serialized form will contain the 'mean' tensor
torch.save(m.state_dict(), 'mean.pt')

m_loaded = RunningMean(4)
m_loaded.load_state_dict(torch.load('mean.pt'))
assert(torch.all(m.mean == m_loaded.mean))
```

As mentioned previously, buffers can be left out of the module's `state_dict` by marking them as non-persistent:

```
self.register_buffer('unserialized_thing', torch.randn(5), persistent=False)
```

Both persistent and non-persistent buffers are affected by model-wide device / dtype changes applied with :func:`~torch.nn.Module.to`:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 455);** *backlink*

Unknown interpreted text role "func".

```
# Moves all module parameters and buffers to the specified device / dtype
m.to(device='cuda', dtype=torch.float64)
```

Buffers of a module can be iterated over using :func:`~torch.nn.Module.buffers` or :func:`~torch.nn.Module.named_buffers`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 463); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 463); *backlink*

Unknown interpreted text role "func".

```
for buffer in m.named_buffers():
  print(buffer)
```

The following class demonstrates the various ways of registering parameters and buffers within a module:

```
class StatefulModule(nn.Module):
  def __init__(self):
    super().__init__()
    # Setting a nn.Parameter as an attribute of the module automatically registers the tensor
    # as a parameter of the module.
    self.param1 = nn.Parameter(torch.randn(2))

    # Alternative string-based way to register a parameter.
    self.register_parameter('param2', nn.Parameter(torch.randn(3)))

    # Reserves the "param3" attribute as a parameter, preventing it from being set to anything
    # except a parameter. "None" entries like this will not be present in the module's state_dict.
    self.register_parameter('param3', None)

    # Registers a list of parameters.
    self.param_list = nn.ParameterList([nn.Parameter(torch.randn(2)) for i in range(3)])

    # Registers a dictionary of parameters.
    self.param_dict = nn.ParameterDict({
      'foo': nn.Parameter(torch.randn(3)),
      'bar': nn.Parameter(torch.randn(4))
    })

    # Registers a persistent buffer (one that appears in the module's state_dict).
    self.register_buffer('buffer1', torch.randn(4), persistent=True)

    # Registers a non-persistent buffer (one that does not appear in the module's state_dict).
    self.register_buffer('buffer2', torch.randn(5), persistent=False)

    # Reserves the "buffer3" attribute as a buffer, preventing it from being set to anything
    # except a buffer. "None" entries like this will not be present in the module's state_dict.
    self.register_buffer('buffer3', None)

    # Adding a submodule registers its parameters as parameters of the module.
    self.linear = nn.Linear(2, 3)

m = StatefulModule()

# Save and load state_dict.
torch.save(m.state_dict(), 'state.pt')
m_loaded = StatefulModule()
m_loaded.load_state_dict(torch.load('state.pt'))

# Note that non-persistent buffer "buffer2" and reserved attributes "param3" and "buffer3" do
# not appear in the state_dict.
print(m_loaded.state_dict())
: OrderedDict([('param1', tensor([-0.0322,  0.9066])),
               ('param2', tensor([-0.4472,  0.1409,  0.4852])),
               ('buffer1', tensor([ 0.6949, -0.1944,  1.2911, -2.1044])),
               ('param_list.0', tensor([ 0.4202, -0.1953])),
               ('param_list.1', tensor([ 1.5299, -0.8747])),
               ('param_list.2', tensor([-1.6289,  1.4898])),
               ('param_dict.bar', tensor([-0.6434,  1.5187,  0.0346, -0.4077])),
```

```
              ('param_dict.foo', tensor([-0.0845, -1.4324,  0.7022])),
              ('linear.weight', tensor([[-0.3915, -0.6176],
                                        [ 0.6062, -0.5992],
                                        [ 0.4452, -0.2843]])),
              ('linear.bias', tensor([-0.3710, -0.0795, -0.3947]))])
```

For more information, check out:

- Saving and loading: https://pytorch.org/tutorials/beginner/saving_loading_models.html
- Serialization semantics: https://pytorch.org/docs/master/notes/serialization.html
- What is a state dict? https://pytorch.org/tutorials/recipes/recipes/what_is_state_dict.html

Module Initialization

By default, parameters and floating-point buffers for modules provided by :mod:`torch.nn` are initialized during module instantiation as 32-bit floating point values on the CPU using an initialization scheme determined to perform well historically for the module type. For certain use cases, it may be desired to initialize with a different dtype, device (e.g. GPU), or initialization technique.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 543); *backlink*
>
> Unknown interpreted text role "mod".

Examples:

```
# Initialize module directly onto GPU.
m = nn.Linear(5, 3, device='cuda')

# Initialize module with 16-bit floating point parameters.
m = nn.Linear(5, 3, dtype=torch.half)

# Skip default parameter initialization and perform custom (e.g. orthogonal) initialization.
m = torch.nn.utils.skip_init(nn.Linear, 5, 3)
nn.init.orthogonal_(m.weight)
```

Note that the device and dtype options demonstrated above also apply to any floating-point buffers registered for the module:

```
m = nn.BatchNorm2d(3, dtype=torch.half)
print(m.running_mean)
: tensor([0., 0., 0.], dtype=torch.float16)
```

While module writers can use any device or dtype to initialize parameters in their custom modules, good practice is to use `dtype=torch.float` and `device='cpu'` by default as well. Optionally, you can provide full flexibility in these areas for your custom module by conforming to the convention demonstrated above that all :mod:`torch.nn` modules follow:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 571); *backlink*
>
> Unknown interpreted text role "mod".

- Provide a `device` constructor kwarg that applies to any parameters / buffers registered by the module.
- Provide a `dtype` constructor kwarg that applies to any parameters / floating-point buffers registered by the module.
- Only use initialization functions (i.e. functions from :mod:`torch.nn.init`) on parameters and buffers within the module's constructor. Note that this is only required to use :func:`~torch.nn.utils.skip_init`; see this page for an explanation.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 579); *backlink*
>
> Unknown interpreted text role "mod".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst, line 579); *backlink*
>
> Unknown interpreted text role "func".

For more information, check out:

- Skipping module parameter initialization: https://pytorch.org/tutorials/prototype/skip_param_init.html

Module Hooks

In :ref:`Neural Network Training with Modules`, we demonstrated the training process for a module, which iteratively performs forward and backward passes, updating module parameters each iteration. For more control over this process, PyTorch provides "hooks" that can perform arbitrary computation during a forward or backward pass, even modifying how the pass is done if desired. Some useful examples for this functionality include debugging, visualizing activations, examining gradients in-depth, etc. Hooks can be added to modules you haven't written yourself, meaning this functionality can be applied to third-party or PyTorch-provided modules.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst`, line 590); *backlink*
>
> Unknown interpreted text role "ref".

PyTorch provides two types of hooks for modules:

- **Forward hooks** are called during the forward pass. They can be installed for a given module with :func:`~torch.nn.Module.register_forward_pre_hook` and :func:`~torch.nn.Module.register_forward_hook`. These hooks will be called respectively just before the forward function is called and just after it is called. Alternatively, these hooks can be installed globally for all modules with the analagous :func:`~torch.nn.modules.module.register_module_forward_pre_hook` and :func:`~torch.nn.modules.module.register_module_forward_hook` functions.

  > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst`, line 599); *backlink*
  >
  > Unknown interpreted text role "func".

  > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst`, line 599); *backlink*
  >
  > Unknown interpreted text role "func".

  > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst`, line 599); *backlink*
  >
  > Unknown interpreted text role "func".

  > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst`, line 599); *backlink*
  >
  > Unknown interpreted text role "func".

- **Backward hooks** are called during the backward pass. They can be installed with :func:`~torch.nn.Module.register_full_backward_hook`. These hooks will be called when the backward for this Module has been computed and will allow the user to access the gradients for both the inputs and outputs. Alternatively, they can be installed globally for all modules with :func:`~torch.nn.modules.module.register_module_full_backward_hook`.

  > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst`, line 605); *backlink*
  >
  > Unknown interpreted text role "func".

  > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)modules.rst`, line 605); *backlink*
  >
  > Unknown interpreted text role "func".

All hooks allow the user to return an updated value that will be used throughout the remaining computation. Thus, these hooks can be used to either execute arbitrary code along the regular module forward/backward or modify some inputs/outputs without having to change the module's `forward()` function.

Below is an example demonstrating usage of forward and backward hooks:

```
torch.manual_seed(1)

def forward_pre_hook(m, inputs):
  # Allows for examination and modification of the input before the forward pass.
  # Note that inputs are always wrapped in a tuple.
  input = inputs[0]
  return input + 1.

def forward_hook(m, inputs, output):
  # Allows for examination of inputs / outputs and modification of the outputs
  # after the forward pass. Note that inputs are always wrapped in a tuple while outputs
  # are passed as-is.

  # Residual computation a la ResNet.
  return output + inputs[0]

def backward_hook(m, grad_inputs, grad_outputs):
  # Allows for examination of grad_inputs / grad_outputs and modification of
  # grad_inputs used in the rest of the backwards pass. Note that grad_inputs and
  # grad_outputs are always wrapped in tuples.
  new_grad_inputs = [torch.ones_like(gi) * 42. for gi in grad_inputs]
  return new_grad_inputs

# Create sample module & input.
m = nn.Linear(3, 3)
x = torch.randn(2, 3, requires_grad=True)

# ==== Demonstrate forward hooks. ====
# Run input through module before and after adding hooks.
print('output with no forward hooks: {}'.format(m(x)))
: output with no forward hooks: tensor([[-0.5059, -0.8158,  0.2390],
                                        [-0.0043,  0.4724, -0.1714]], grad_fn=<AddmmBackward>)

# Note that the modified input results in a different output.
forward_pre_hook_handle = m.register_forward_pre_hook(forward_pre_hook)
print('output with forward pre hook: {}'.format(m(x)))
: output with forward pre hook: tensor([[-0.5752, -0.7421,  0.4942],
                                        [-0.0736,  0.5461,  0.0838]], grad_fn=<AddmmBackward>)

# Note the modified output.
forward_hook_handle = m.register_forward_hook(forward_hook)
print('output with both forward hooks: {}'.format(m(x)))
: output with both forward hooks: tensor([[-1.0980,  0.6396,  0.4666],
                                          [ 0.3634,  0.6538,  1.0256]], grad_fn=<AddBackward0>)

# Remove hooks; note that the output here matches the output before adding hooks.
forward_pre_hook_handle.remove()
forward_hook_handle.remove()
print('output after removing forward hooks: {}'.format(m(x)))
: output after removing forward hooks: tensor([[-0.5059, -0.8158,  0.2390],
                                               [-0.0043,  0.4724, -0.1714]], grad_fn=<AddmmBackward>)

# ==== Demonstrate backward hooks. ====
m(x).sum().backward()
print('x.grad with no backwards hook: {}'.format(x.grad))
: x.grad with no backwards hook: tensor([[ 0.4497, -0.5046,  0.3146],
                                         [ 0.4497, -0.5046,  0.3146]])

# Clear gradients before running backward pass again.
m.zero_grad()
x.grad.zero_()

m.register_full_backward_hook(backward_hook)
m(x).sum().backward()
print('x.grad with backwards hook: {}'.format(x.grad))
: x.grad with backwards hook: tensor([[42., 42., 42.],
                                      [42., 42., 42.]])
```

Advanced Features

PyTorch also provides several more advanced features that are designed to work with modules. All these functionalities are available for custom-written modules, with the small caveat that certain features may require modules to conform to particular constraints in order to be supported. In-depth discussion of these features and the corresponding requirements can be found in the links below.

Distributed Training

Various methods for distributed training exist within PyTorch, both for scaling up training using multiple GPUs as well as training across multiple machines. Check out the distributed training overview page for detailed information on how to utilize these.

Profiling Performance

The PyTorch Profiler can be useful for identifying performance bottlenecks within your models. It measures and outputs performance characteristics for both memory usage and time spent.

Improving Performance with Quantization

Applying quantization techniques to modules can improve performance and memory usage by utilizing lower bitwidths than floating-point precision. Check out the various PyTorch-provided mechanisms for quantization here.

Improving Memory Usage with Pruning

Large deep learning models are often over-parametrized, resulting in high memory usage. To combat this, PyTorch provides mechanisms for model pruning, which can help reduce memory usage while maintaining task accuracy. The Pruning tutorial describes how to utilize the pruning techniques PyTorch provides or define custom pruning techniques as necessary.

Deploying with TorchScript

When deploying a model for use in production, the overhead of Python can be unacceptable due to its poor performance characteristics. For cases like this, TorchScript provides a way to load and run an optimized model program from outside of Python, such as within a C++ program.

Parametrizations

For certain applications, it can be beneficial to constrain the parameter space during model training. For example, enforcing orthogonality of the learned parameters can improve convergence for RNNs. PyTorch provides a mechanism for applying parametrizations such as this, and further allows for custom constraints to be defined.

Transforming Modules with FX

The FX component of PyTorch provides a flexible way to transform modules by operating directly on module computation graphs. This can be used to programmatically generate or manipulate modules for a broad array of use cases. To explore FX, check out these examples of using FX for convolution + batch norm fusion and CPU performance analysis.