MODULE

CLASS `torch.nn.Module` [SOURCE]

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
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))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

`add_module(name: str, module: Optional[Module])` → None [SOURCE]

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*string*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

`apply(fn: Callable[Module, None])` → T [SOURCE]

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also torch.nn.init).

Parameters

fn (`Module` -> None) – function to be applied to each submodule

Returns

self

Return type

Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

`bfloat16`() → T [SOURCE]

Casts all floating point parameters and buffers to `bfloat16` datatype.

Returns

self

Return type

Module

`buffers`(*recurse: bool = True*) → Iterator[torch.Tensor] [SOURCE]

Returns an iterator over module buffers.

Parameters

recurse (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields

torch.Tensor – module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

`children`() → Iterator[torch.nn.modules.module.Module] [SOURCE]

Returns an iterator over immediate children modules.

Yields

Module – a child module

`cpu`() → T [SOURCE]

Moves all model parameters and buffers to the CPU.

Returns

self

Return type

Module

`cuda(device: Union[int, torch.device, None] = None) → T` [SOURCE]

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Parameters

device (*int, optional*) – if specified, all parameters will be copied to that device

Returns

self

Return type

Module

`double() → T` [SOURCE]

Casts all floating point parameters and buffers to `double` datatype.

Returns

self

Return type

Module

`dump_patches: BOOL = FALSE`

This allows better BC support for `load_state_dict()`. In `state_dict()`, the version number will be saved as in the attribute *_metadata* of the returned state dict, and thus pickled. *_metadata* is a dictionary with keys that follow the naming convention of state dict. See `_load_from_state_dict` on how to use this information in loading.

If new parameters/buffers are added/removed from a module, this number shall be bumped, and the module's *_load_from_state_dict* method can compare the version number and do appropriate changes if the state dict is from before the change.

`eval() → T` [SOURCE]

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

Returns

self

Return type

Module

`extra_repr() → str` [SOURCE]

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

`float() → T` [SOURCE]

Casts all floating point parameters and buffers to float datatype.

Returns

self

Return type

Module

`forward`(**input: Any*) → None

Defines the computation performed at every call.

Should be overridden by all subclasses.

- NOTE

  Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`half`() → T [SOURCE]

Casts all floating point parameters and buffers to `half` datatype.

Returns

self

Return type

Module

`load_state_dict`(*state_dict: Dict[str, torch.Tensor], strict: bool = True*) [SOURCE]

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type

`NamedTuple` with `missing_keys` and `unexpected_keys` fields

`modules`() → Iterator[torch.nn.modules.module.Module] [SOURCE]

Returns an iterator over all modules in the network.

Yields

Module – a module in the network

- NOTE

  Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

`named_buffers`(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]] [SOURCE]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields

(*string, torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

`named_children`() → Iterator[Tuple[str, torch.nn.modules.module.Module]] [SOURCE]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields

(*string, Module*) – Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

`named_modules`(*memo: Optional[Set[Module]] = None*, *prefix: str = ''*) [SOURCE]

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields

(*string, Module*) – Tuple of name and module

- NOTE

Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

`named_parameters`(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]] [SOURCE]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields

(*string, Parameter*) – Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

`parameters`(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter] [SOURCE]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters

recurse (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields

Parameter – module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

`register_backward_hook`(*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle [SOURCE]

Registers a backward hook on the module.

- WARNING

  The current implementation will not have the presented behavior for complex `Module` that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such `Module`, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemovableHandle`

`register_buffer`(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → None [SOURCE]

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of

the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*string*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor*) – buffer to be registered.
- **persistent** (*bool*) – whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

`register_forward_hook(hook: Callable[..., None])` → torch.utils.hooks.RemovableHandle [SOURCE]

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemovableHandle`

`register_forward_pre_hook(hook: Callable[..., None])` → torch.utils.hooks.RemovableHandle [SOURCE]

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemovableHandle`

`register_parameter(name: str, param: Optional[torch.nn.parameter.Parameter])` → None [SOURCE]

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*string*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter*) – parameter to be added to the module.

`requires_grad_(requires_grad: bool = True)` → T [SOURCE]

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Parameters

requires_grad (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns

self

Return type

Module

`state_dict(destination=None, prefix='', keep_vars=False)` [SOURCE]

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns

a dictionary containing a whole state of the module

Return type

dict

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

`to(*args, **kwargs)` [SOURCE]

Moves and/or casts the parameters and buffers.

This can be called as

`to(device=None, dtype=None, non_blocking=False)` [SOURCE]

`to(dtype, non_blocking=False)` [SOURCE]

`to(tensor, non_blocking=False)` [SOURCE]

`to(memory_format=torch.channels_last)` [SOURCE]

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype` s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

- NOTE

  This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns

self

Return type

Module

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

`train`(*mode: bool = True*) → T [SOURCE]

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters

mode (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns

self

Return type

Module

`type`(*dst_type: Union[torch.dtype, str]*) → T [SOURCE]

Casts all parameters and buffers to `dst_type`.

Parameters

dst_type (*type or string*) – the desired type

Returns

self

Return type

Module

`zero_grad`(*set_to_none: bool = False*) → None [SOURCE]

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters

set_to_none (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

Previous Next