深入剖析Pytorch的nn.Module源码

2022.07.22

本文是对nn.Module中的常用函数源码进行剖析（Module在pytorch中是大部分类的基类）

1.init函数

包含很多成员变量，一般是字典格式，默认情况下shuffle、dropout都是遵循training=true设置的

```
    def __init__(self) -> None:
        """
        Initializes internal Module state, shared by both nn.Module and
ScriptModule.
        """
        torch._C._log_api_usage_once("python.nn_module")

        self.training = True # 设置 training = True 训练模式
        self._parameters: Dict[str, Optional[Parameter]] = OrderedDict()
        self._buffers: Dict[str, Optional[Tensor]] = OrderedDict()
        self._non_persistent_buffers_set: Set[str] = set()
        self._backward_hooks: Dict[int, Callable] = OrderedDict()
        self._is_full_backward_hook = None
        self._forward_hooks: Dict[int, Callable] = OrderedDict()
        self._forward_pre_hooks: Dict[int, Callable] = OrderedDict()
        self._state_dict_hooks: Dict[int, Callable] = OrderedDict()
        self._load_state_dict_pre_hooks: Dict[int, Callable] = OrderedDict()
        self._load_state_dict_post_hooks: Dict[int, Callable] = OrderedDict()
        self._modules: Dict[str, Optional['Module']] = OrderedDict()
```

2.register_buffer

作用：往当前模型中添加buffer。一般我们不能将buffer视为模型的参数，默认情况下buffers是持久的，可以和parameters一起保存，当然也可以设置False，就不会被保存了。参数说明：

- name：buffer名称
- tensor：注册的buffer张量的值
- persistent：是否作为张量保存下来

```
def register_buffer(self, name: str, tensor: Optional[Tensor], persistent: bool
= True) -> None:
    pass # 具体的逻辑判断
```

3.register_parameter

作用：往模型中添加参数，使用频率较高。参数说明：

name：字符串形式，添加参数的名称

parameter：是tensor形式的继承，但必须写成Parameter（一个类）的实例形式，而不是简单的一个tensor

```
def register_parameter(self, name: str, param: Optional[Parameter]) -> None:
    pass # 具体的逻辑判断
```

使用举例：https://www.codenong.com/cs106951116/

```
class Example(nn.Module):
    def __init__(self):
        super(Example, self).__init__()
        print('看看我们的模型有哪些parameter:\t', self._parameters, end='\n')
        self.W1_params = nn.Parameter(torch.rand(2,3))
        print('增加W1后看看：',self._parameters, end='\n')

        self.register_parameter('W2_params' , nn.Parameter(torch.rand(2,3))) #
register parameter
        print('增加W2后看看：',self._parameters, end='\n')
    def forward(self, x):
        return x
```

4.add_module

作用：往当前module中添加子模块。参数说明：

name：添加子模块的名称

module：Module类的实例

```
def add_module(self, name: str, module: Optional['Module']) -> None:
    pass
```

5.get_parameter

作用：根据传入的target字符串获取参数，返回的是torch.nn.parameter的实例

```
def get_parameter(self, target: str) -> "Parameter":
        module_path, _, param_name = target.rpartition(".") # 调用rpartition，解析
出module位置与参数名称

        mod: torch.nn.Module = self.get_submodule(module_path)

        if not hasattr(mod, param_name): # 判断mod是否有name属性
            raise AttributeError(mod._get_name() + " has no attribute `"
                                 + param_name + "`")

        param: torch.nn.Parameter = getattr(mod, param_name)

        if not isinstance(param, torch.nn.Parameter):
            raise AttributeError("`" + param_name + "` is not an "
                                 "nn.Parameter") # 判断是否是torch.nn.parameter的实
例

        return param
```

6._save_to_state_dict

作用：把模型所有的buffer、parameters放到一个dict中

```
def _save_to_state_dict(self, destination, prefix, keep_vars):
        for name, param in self._parameters.items(): # 遍历参数和
buffer,_parameters只是当前模块的，不对子模块遍历
            if param is not None:
                destination[prefix + name] = param if keep_vars else
param.detach()
        for name, buf in self._buffers.items():
            if buf is not None and name not in self._non_persistent_buffers_set:
                destination[prefix + name] = buf if keep_vars else buf.detach()
        extra_state_key = prefix + _EXTRA_STATE_KEY_SUFFIX
        if getattr(self.__class__, "get_extra_state", Module.get_extra_state) is
not Module.get_extra_state:
            destination[extra_state_key] = self.get_extra_state()
```

7.state_dict

作用：对当前module和子module进行递归计算，把他们的buffer和参数都存储到destination这个字典中，最终返回这样的一个字典

```
def state_dict(self, *args, destination=None, prefix='', keep_vars=False):
        if len(args) > 0:
            if destination is None:
                destination = args[0]
            if len(args) > 1 and prefix == '':
                prefix = args[1]
            if len(args) > 2 and keep_vars is False:
                keep_vars = args[2]
            # DeprecationWarning is ignored by default
            warnings.warn(
                "Positional args are being deprecated, use kwargs instead. Refer
to "

 "https://pytorch.org/docs/master/generated/torch.nn.Module.html#torch.nn.Module
.state_dict"
                " for details.")

        if destination is None:
            destination = OrderedDict()
            destination._metadata = OrderedDict()

        local_metadata = dict(version=self._version)
        if hasattr(destination, "_metadata"):
            destination._metadata[prefix[:-1]] = local_metadata

        self._save_to_state_dict(destination, prefix, keep_vars) # 第一步，把当前模
块的参数和buffer存储到字典destination中
        for name, module in self._modules.items(): # 对子模块进行遍历
            if module is not None:
                module.state_dict(destination=destination, prefix=prefix + name
+ '.', keep_vars=keep_vars)
        for hook in self._state_dict_hooks.values():
```

```
            hook_result = hook(self, destination, prefix, local_metadata)
            if hook_result is not None:
                destination = hook_result
        return destination
```

8.load_state_dict

作用：从原先存储的字典中导入（获取）当前模块的参数和buffer，然后对所有子模块都会这样遍历

9.区分 _parameters、parameters()、named_parameters()

- _parameters返回的是当前模块（不包含子模块）的参数
- 调用parameters()可以返回当前模块和子模块的参数
- named_parameters将返回模块名称以及对应的参数值，更加清晰一点

10.区分 _modules 、named_modules()、 named_children()、modules()

- _modules返回的是一个字典，键是module的名称，值是module的具体情况（不包含自身，只有子模块）
- named_modules()返回了包含了自身的所有模块，含名称和具体情况
- named_children()返回的是一个个元组，是每个子模块，并且写明了名称
- modules()返回的是每个模块的具体情况，但没有名称

11.train

```
def train(self: T, mode: bool = True) -> T:
        if not isinstance(mode, bool): # 判断是否是bool类型
            raise ValueError("training mode is expected to be boolean")
        self.training = mode # 设置True or False
        for module in self.children(): # 对子模块也如此操作
            module.train(mode)
        return self
```

当设置 train = True or False 后，相应的，它的子模块也会相应地设置成True or False，那么就可能会影响到一些类的使用，比如 Dropout（因为dropout相当于它的子模块），dropout的运行逻辑就会发生变化（相应地，进入训练或验证模式）：

```
class Dropout(_DropoutNd): # 这里的 _DropoutNd 继承自 Module 类
    def forward(self, input: Tensor) -> Tensor:
        return F.dropout(input, self.p, self.training, self.inplace) #
self.training
```

12.eval

进入验证推理模型

```
def eval(self: T) -> T:
    return self.train(False) # 调用train函数，并传入False
```

总结来说，只要在一个大模型中设置train=Ture or False就行，不需要额外设置dropout、BN之类的