**Note Information**

Author : AbbasXu
Date : 2022-08-18
Title : pytorch基础
Keywords : #pytroch #模型搭建 #训练

---

# 模型定义的方式

## 回顾

- `Module` 类是 `torch.nn` 模块里提供的一个模型构造类 (`nn.Module`)，是所有神经网网络模块的基类，我们可以继承它来定义我们想要的模型；
- PyTorch模型定义应包括两个主要部分：各个部分的初始化（`__init__`）；数据流向定义（`forward`）
- 基于nn.Module，我们可以通过Sequential，ModuleList和ModuleDict三种方式定义PyTorch模型。

## Sequential

- 使用场景：当模型的前向计算为简单串联各个层的计算时
- 排列方式：
  - 直接排列

```
net = nn.Sequential(
        nn.Linear(784, 256),
        nn.ReLU(),
        nn.Linear(256, 10),
        )
```

  - 使用OrderedDict

```
net2 = nn.Sequential(collections.OrderedDict([
          ('fc1', nn.Linear(784, 256)),
          ('relu1', nn.ReLU()),
          ('fc2', nn.Linear(256, 10))
          ]))
```

- 特点：简单、易读且不需要写forward函数；但同时不够灵活

## ModuleList

- 特点：可以像list那样进行append和extend操作，但该方法不会定义一个网络，需要使用forward函数对顺序进行确定

## ModuleDict

- 特点：ModuleDict和ModuleList的作用类似，只是ModuleDict能够更方便地为神经网络的层添加名称。

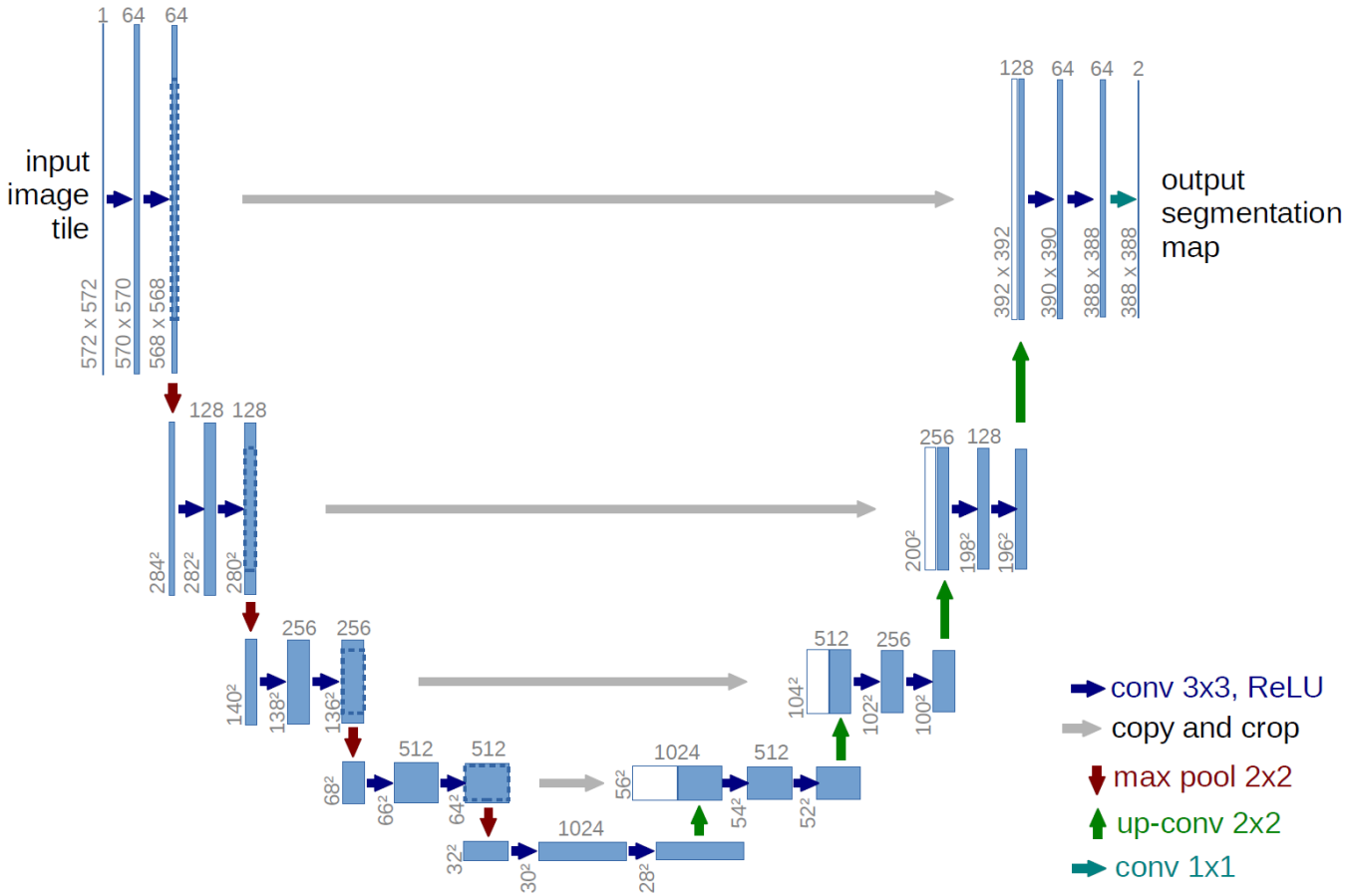
注：ModuleList和ModuleDict的使用场景为某个完全相同的层需要重复出现多次时，非常方便实现，可以”一行顶多行“

---

# 利用模型块快速搭建复杂网络

## U-net简介

U-Net是分割 (Segmentation) 模型的杰作，在以医学影像为代表的诸多领域有着广泛的应用。U-Net模型结构如下图所示，通过残差连接结构解决了模型学习中的退化问题，使得神经网络的深度能够不断扩展。



组成U-Net的模型块主要有如下几个部分：

- 每个子块内部的两次卷积（Double Convolution）
- 左侧模型块之间的下采样连接，即最大池化（Max pooling）
- 右侧模型块之间的上采样连接（Up sampling）
- 输出层的处理

## U-Net模型块实现

针对上述的四个部分，分别实现一个Class，下面以DoubleConv为例：

```
class DoubleConv(nn.Module):
    """(convolution => [BN] => ReLU) * 2"""

    def __init__(self, in_channels, out_channels, mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, kernel_size=3, padding=1,
bias=False),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_channels, out_channels, kernel_size=3, padding=1,
bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.double_conv(x)
```

## 利用模型块组装U-Net

使用写好的模型块，可以非常方便地组装U-Net模型。可以看到，通过模型块的方式实现了代码复用，整个模型结构定义所需的代码总行数明显减少，代码可读性也得到了提升。

```
class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=False):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inc = DoubleConv(n_channels, 64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        factor = 2 if bilinear else 1
        self.down4 = Down(512, 1024 // factor)
        self.up1 = Up(1024, 512 // factor, bilinear)
        self.up2 = Up(512, 256 // factor, bilinear)
        self.up3 = Up(256, 128 // factor, bilinear)
        self.up4 = Up(128, 64, bilinear)
        self.outc = OutConv(64, n_classes)

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        logits = self.outc(x)
        return logits
```

# Pytorch修改模型

## 修改模型层

以模型ResNet50为例

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
..............
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

这里模型结构是为了适配ImageNet预训练的权重，因此最后全连接层（fc）的输出节点数是1000。

假设我们要用这个resnet模型去做一个10分类的问题，就应该修改模型的fc层，将其输出节点数替换为10。另外，我们觉得一层全连接层可能太少了，想再加一层。可以做如下修改：

```
from collections import OrderedDict
classifier = nn.Sequential(OrderedDict([('fc1', nn.Linear(2048, 128)),
                          ('relu1', nn.ReLU()),
                          ('dropout1',nn.Dropout(0.5)),
                          ('fc2', nn.Linear(128, 10)),
                          ('output', nn.Softmax(dim=1))
                          ]))

net.fc = classifier
```

这里的操作相当于将模型（net）最后名称为“fc”的层替换成了名称为“classifier”的结构，该结构是我们自己定义的。

## 添加外部输入

使用场景：在模型训练中，除了已有模型的输入之外，还需要输入额外的信息。
即将原模型添加输入位置前的部分作为一个整体，同时在forward中定义好原模型不变的部分、添加的输入和后续层之间的连接关系，从而完成模型的修改。
以torchvision的resnet50模型为基础，在倒数第二层增加一个额外的输入变量add_variable来辅助预测。

```
class Model(nn.Module):
    def __init__(self, net):
        super(Model, self).__init__()
        self.net = net
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc_add = nn.Linear(1001, 10, bias=True)
        self.output = nn.Softmax(dim=1)

    def forward(self, x, add_variable):
        x = self.net(x)
        x = torch.cat((self.dropout(self.relu(x)),
add_variable.unsqueeze(1)),1)
        x = self.fc_add(x)
        x = self.output(x)
        return x
```

这里对外部输入变量"add_variable"进行unsqueeze操作（作用是用于增加维度，操作是针对于tensor张量，增加一个维数为1的维度。）是为了和net输出的tensor保持维度一致，常用于

add_variable是单一数值 (scalar) 的情况，此时add_variable的维度是 (batch_size, )，需要在第二维补充维数1，从而可以和tensor进行torch.cat操作。

### 添加额外输出

使用场景：在模型训练中，除了模型最后的输出外，我们需要输出模型某一中间层的结果，以施加额外的监督，获得更好的中间层结果。
基本的思路是修改模型定义中forward函数的return变量。
以torchvision的resnet50模型为例，同时输出1000维的倒数第二层和10维的最后一层结果。

```
class Model(nn.Module):
    def __init__(self, net):
        super(Model, self).__init__()
        self.net = net
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc1 = nn.Linear(1000, 10, bias=True)
        self.output = nn.Softmax(dim=1)

    def forward(self, x, add_variable):
        x1000 = self.net(x)
        x10 = self.dropout(self.relu(x1000))
        x10 = self.fc1(x10)
        x10 = self.output(x10)
        return x10, x1000
```

## PyTorch模型保存与读取

### 模型存储格式

- pkl
- pt（官方文档常用）
- pth

### 模型存储内容

两种方案：

- 保存整个模型（包括权重和模型结构）
- 仅保存模型权重

## 单卡和多卡模型存储的区别

如果要使用多卡训练的话，需要对模型使用torch.nn.DataParallel。

## 情况分类讨论

目前和可预想的未来一段时间内我只接触到单卡保存和单卡训练的情况，因此仅记录该情况。

```
import os
import torch
from torchvision import models

os.environ['CUDA_VISIBLE_DEVICES'] = '0'   #这里替换成希望使用的GPU编号
model = models.resnet152(pretrained=True)
model.cuda()

# 保存+读取整个模型
torch.save(model, save_dir)
loaded_model = torch.load(save_dir)
loaded_model.cuda()

# 保存+读取模型权重
torch.save(model.state_dict(), save_dir)
loaded_dict = torch.load(save_dir)
loaded_model = models.resnet152()   #注意这里需要对模型结构有定义
loaded_model.state_dict = loaded_dict
loaded_model.cuda()
```