网络构建指南

Pytorch中模型训练步骤还是非常清晰的：

数据载入及处理

模型定义

超参数设置（损失函数定义、优化器定义、训练轮数）

训练模型

读取一个batch的数据，并前向传播
计算损失值
反向传播计算梯度
优化器优化模型
循环执行上述过程直到规定轮数
评估模型（非必须）

测试模型

其中除了损失函数和优化器的定义和使用没有提到，其余内容在前文都有介绍，下面直接搭建一个CNN网络，展示一个网络的完整训练流程：

```

"""
依赖包载入、数据集载入和划分
以CIFAR10作为模型训练的数据集，训练集50000张，测试集10000张图片
"""
import torchvision
import torch.nn as nn
import torch
from torch.utils.data import DataLoader
from torchvision import transforms, datasets

transform = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ]
)
# 准备数据集
train_data = datasets.CIFAR10(root="./dataset", train=True, transform=transform, download=True)
test_data = datasets.CIFAR10(root="./dataset", train=False, transform=transform, download=True)

# length
train_data_size = len(train_data)
```

```
test_data_size = len(test_data)
print("训练数据集长度为：{} \n验证数据集的长度为：{}".format(train_data_size, test_data_size))



# 利用DataLoader加载数据集
train_dataloader = DataLoader(train_data, shuffle=True,batch_size=32, num_workers=15)
test_dataloader = DataLoader(test_data, shuffle=False,batch_size=10000, num_workers= 15)

# test_iter = iter(test_dataloader)
# test_imgs, test_labels = test_iter.next()
# test_imgs.shape
# test_imgs = test_imgs.to(device)
# test_labels = test_labels.to(device)

Files already downloaded and verified
Files already downloaded and verified
训练数据集长度为：50000
验证数据集的长度为：10000

"""
搭建LeNet网络
"""
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(3, 16, 5),  # input_size:[3,32,32]     out_size: [16,28,28]
            nn.Sigmoid(),
            nn.AvgPool2d(2),  # input_size: [16,28,28]   out_size: [16,14,14]
            nn.Conv2d(16, 32, 5),  # input_size: [16,14,14]    out_size: [32,10,10]
            nn.Sigmoid(),
            nn.AvgPool2d(2),  # input_size: [32,10,10]   out_size: [32,5,5]
            nn.Flatten(),  # 矩阵展开
            nn.Linear(32 * 5 * 5, 120), nn.Sigmoid(),
            nn.Linear(120, 84), nn.Sigmoid(),
            nn.Linear(84, 10)
        )

    def forward(self, x):
        x = self.model(x)
        return x


"""
训练模型
"""
from tqdm import tqdm
import sys
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = LeNet()
model = model.to(device)  # 设置在GPU中训练
```

```python

# 损失函数
loss_fn = nn.CrossEntropyLoss().to(device)

# 优化器
learning_rate = 0.005
optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)

# 设置训练网络的参数
epochs = 5

# 添加tensorboard
#writer = SummaryWriter("./logs_train_CIFAR10")
# 开始训练
best= 0
for epoch in range(epochs):
    print("-------第 {} 轮训练开始-------".format(epoch+1))
    train_bar = tqdm(train_dataloader, file=sys.stdout)
    model.train() #网络中有特殊层的时候需要加上，具体看文档，但加上不会出错
    running_loss = 0.0
    for step,data in enumerate(train_bar):
        imgs, targets = data
        imgs = imgs.to(device)
        targets = targets.to(device)
        outputs = model(imgs)
        loss = loss_fn(outputs, targets)

        # 优化器优化模型
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

        train_bar.desc = "train epoch[{}/{}] loss:{:.3f}".format(epoch + 1,epochs, loss)

    # 测试步骤开始
    model.eval() # 网络中有特殊层的时候需要加上，具体看文档，但加上不会出错
    total_test_loss = 0
    total_accuracy = 0
    with torch.no_grad(): # 取消梯度跟踪，进行测试 重要！！！
        for data in test_dataloader:
            imgs, targets = data
            imgs = imgs.to(device)
            targets = targets.to(device)
            outputs = model(imgs)
            print(outputs.shape)
            loss = loss_fn(outputs, targets)
            total_test_loss = total_test_loss + loss.item()
            accurcy = (torch.max(outputs, dim=1)[1] == targets).sum().item()
            total_accuracy = total_accuracy + accurcy

    print('[epoch %d] train_loss: %.3f  val_accuracy: %.3f' %
              (epoch + 1, running_loss / len(train_dataloader), total_accuracy/test_data_size))
```

```
    if best < total_accuracy/test_data_size:
        best = total_accuracy/test_data_size
        torch.save(model.state_dict(), "./Model/LeNet_{}.path".format(epochs))

# 保存每一次训练的模型
print("------训练完毕-------")
# writer.close()

# -------第 1 轮训练开始-------
# train epoch[1/5] loss:1.944: 100%|██████████| 1563/1563 [00:12<00:00, 121.29it/s]
# torch.Size([10000, 10])
# [epoch 1] train_loss: 2.022  val_accuracy: 0.291
# -------第 2 轮训练开始-------
# train epoch[2/5] loss:1.620: 100%|██████████| 1563/1563 [00:12<00:00, 121.47it/s]
# torch.Size([10000, 10])
# [epoch 2] train_loss: 1.761  val_accuracy: 0.380
# -------第 3 轮训练开始-------
# train epoch[3/5] loss:1.420: 100%|██████████| 1563/1563 [00:12<00:00, 127.57it/s]
# torch.Size([10000, 10])
# [epoch 3] train_loss: 1.629  val_accuracy: 0.422
# -------第 4 轮训练开始-------
# train epoch[4/5] loss:1.791: 100%|██████████| 1563/1563 [00:12<00:00, 123.46it/s]
# torch.Size([10000, 10])
# [epoch 4] train_loss: 1.551  val_accuracy: 0.428
# -------第 5 轮训练开始-------
# train epoch[5/5] loss:1.358: 100%|██████████| 1563/1563 [00:12<00:00, 121.48it/s]
# torch.Size([10000, 10])
# [epoch 5] train_loss: 1.493  val_accuracy: 0.458
# ------训练完毕-------
```

模型搭建 torch.nn

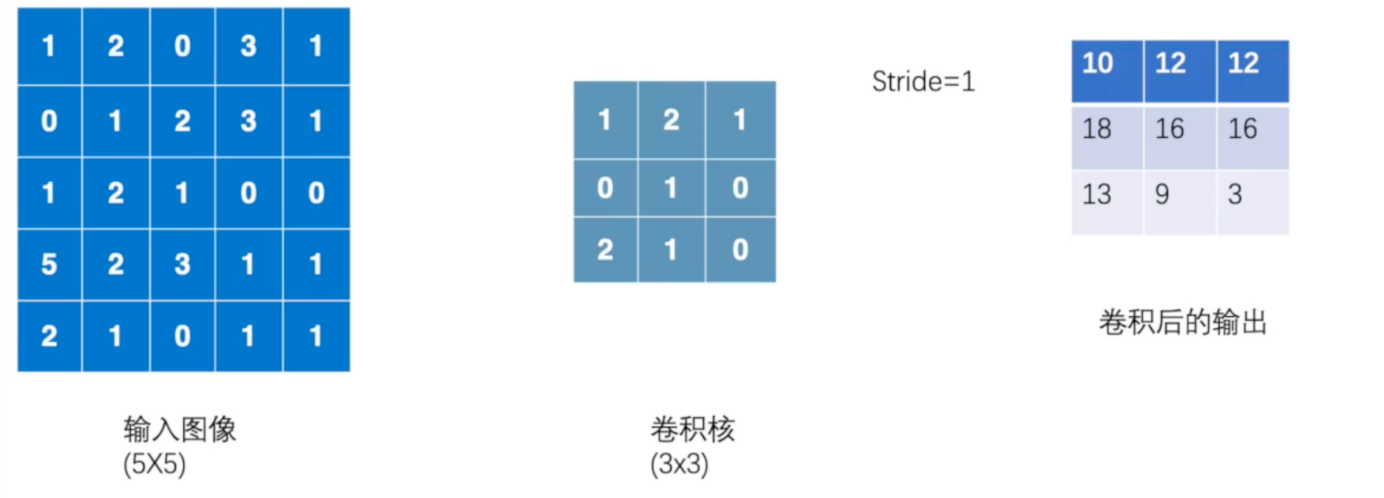
nn全称为neural network，意思是神经网络，是torch中构建神经网络的模块

torch.nn.functional

该模块包含构建神经网络需要的函数，包括卷积层、池化层、激活函数、损失函数、全连接函数等，具体查看官方文档：https://pytorch.org/docs/stable/nn.functional.html#convolution-functions

注意这个模块中只包含了函数，所谓函数就是输入数据得到对应的输出，只是简单的数学运算，没有自动更新权重的能力，与后面介绍的Modules不太一样。

卷积操作举例如下：



下面用`torch.nn.functional.conv2d`模拟一下上图的卷积操作：

需要注意的是，在Pytorch中，只要是nn下的包都只支持 mini-batch ，即输入和输出的数据是4维的，每一维度分别表示：（batch大小，输入通道数，高度，宽度），即`N*C*H*W`，即使只有一张单通道的黑白图片，也要转变为`1*1*H*W`的形式。

> `torch.nn` only supports mini-batches. The entire `torch.nn` package only supports inputs that are a mini-batch of samples, and not a single sample.
>
> > For example, `nn.Conv2d` will take in a 4D Tensor of `nSamples * nChannels * Height * Width`.
> > If you have a single sample, just use `input.unsqueeze(0)` to add a fake batch dimension.

CNN的基本层

Convolution Layers

详情见官方文档：https://pytorch.org/docs/stable/nn.html#convolution-layers

Pytorch中实现了很多常用的卷积层，对于图像处理的卷积神经网络来说，最常用的就是 `nn.Conv2d`，即二维卷积层，这里也以此为例。

`nn.Conv2d` 也是一个类，继承自 `_ConvNd` ，而 `_ConvNd` 又继承自 `Module`。

声明时主要参数：
`torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)`

- **in_channels** (int) – 输入图像的通道数
- **out_channels** (int) – 卷积层输出通道数
- **kernel_size** (int *or* tuple) – 卷积核大小
- **stride** (int *or* tuple, *optional*) – 卷积核步长
- **padding** (int, tuple *or* str, *optional*) – 在外层填充圈数，Default: 0
- **dilation** (int *or* tuple, *optional*) – 卷积核中挖洞，Default: 1（表示不挖洞）

- **bias** (bool, *optional*) – 是否添加偏置项 Default: `True`

其中输入图像和输出图像的大小计算方式如下图：

- Input: $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in})
- Output: $(N, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$, where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

```
import torch.nn as nn
# With square kernels and equal stride
conv = nn.Conv2d(16, 33, 3, stride=2, padding=2)
input = torch.randn(20, 16, 50, 100)
print(input.shape)
output = conv(input) ## __call__实习对象函数式调用
output.shape

```

Pooling layers

具体见官方文档：https://pytorch.org/docs/stable/nn.html#pooling-layers

实现了常用的池化层，如最大池化，平均池化等。以 `nn.MaxPool2d` 为例：

池化层主要是用于缩小图片的维度，减少冗余特征，从而加快训练速度，经过池化层处理后的图像一般通道数不变，只会改变长宽。

主要参数如下：

- kernel_size – the size of the window to take a max over
- stride – the stride of the window. Default value is kernel_size
- padding – implicit zero padding to be added on both sides
- dilation – a parameter that controls the stride of elements in the window

参数含义类似卷积层，不详细解释，输入输出图像的长宽计算公式：

- Input: (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}), where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 * \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 * \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

```
# pool of square window of size=3, stride=2
MaxPool2d = nn.MaxPool2d(3, stride=2)
input = torch.randn(20, 16, 50, 32)
output = MaxPool2d(input)
output.shape

```

搭建一个简易CNN

我们在定义自已的网络的时候，需要继承 nn.Module 类，并重新实现构造函数 __init__ 构造函数和 forward 这两个方法。继承 nn.Module 类在自定义类时即可实现，注意在构造函数中也需要先调用父类的构造函数，forward 接受输入进行前向传播后返回输出结果，由于model类实现了 __call__ ，所以可以直接使用 对象名() 的方式进行前向传播.

在实现__init__ 和 forward 时有一些注意技巧：

（1）一般把网络中具有可学习参数的层（如全连接层、卷积层等）放在构造函数 __init__() 中，当然我也可以把不具有参数的层也放在里面；

（2）一般把不具有可学习参数的层(如ReLU、dropout、BatchNormanation层)可放在构造函数中，也可不放在构造函数中，如果不放在构造函数__init__里面，则在 forward 方法里面可以使用 nn.functional 来代替。

（3）forward 方法是必须要重写的，它是实现模型的功能，实现各个层之间的连接关系的核心。

是否将不具有参数的层放入构造函数的区别在于，只有在构造函数中的层才属于模型的层，其参数才会在训练时被更新，而有些层本来就没有参数无需训练，所以可以不用放在构造函数内，只要在 forward 中实现即可

```
import torch
import torch.nn.functional as F

class MyNet(torch.nn.Module):
    def __init__(self):
        super(MyNet, self).__init__()  # 第一句话，调用父类的构造函数
        self.conv1 = torch.nn.Conv2d(3, 32, 3, 1, 1)
        self.conv2 = torch.nn.Conv2d(3, 32, 3, 1, 1)

        self.dense1 = torch.nn.Linear(32 * 3 * 3, 128)
        self.dense2 = torch.nn.Linear(128, 10)
```

```python

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = F.max_pool2d(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x)
        x = self.dense1(x)
        x = self.dense2(x)
        return x

model = MyNet()
print(model)
```

上采样upconv

```python
nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
```

用法类似