组成部件

同样的，看文档时，关注输入输出的形状，以及必须传入的参数

nn.Module

`nn.Module` (Module — PyTorch 2.7 文档 - PyTorch 深度学习库) 是所有神经网络模块的基类，我们定义的模型都要继承该类，因此可以将其理解为所有神经网络的基本骨架。我们把这个骨架拿来使用，同时对不满意的部分进行修改。

继承 `nn.Module` 后，必须在 `__init__` 函数的第一句调用 `super`，并重写 `forward` 函数。

PyTorch 中的大多数层（如 `nn.Linear`）默认支持批量处理，会自动处理批量输入。通常 batch 大小在第一个维度，但也要看具体层的情况（文档）

使用示例：

PYTHON

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        x += 1
        return x

model = Model()
x = torch.tensor(1.0)
print(f"input: {x}")
output = model(x)
print(f"output: {output}")
```

卷积层

卷积层的理论和数学介绍见CNN，这里只讲代码怎么写。

最常用的是 `Conv2d` (torch.nn — PyTorch 2.7 文档 - PyTorch 深度学习库)，其他的不常用 (Resnet 里会用到 `ConvTranspose2d`)。

需要关注的参数只有前几个，它们作用的可视化可见 conv_arithmetic/README.md at master · vdumoulin/conv_arithmetic · GitHub。 示例代码：

PYTHON

```python
import torch
import torch.nn as nn
import torchvision
from torch.utils.data import DataLoader, Dataset
from torch.utils.tensorboard import SummaryWriter

class MyConv(nn.Module):
    def __init__(self):
        super(MyConv, self).__init__()
        # kernel 只要指定 kernel_size 即可，内部参数值按一定的分布初始化
        # 常用参数就这 5 个
        self.conv = nn.Conv2d(in_channels=3, out_channels=6,
kernel_size=3, stride=1, padding=0)
    def forward(self, x):
        x = self.conv(x)
        # 输出维度可以使用公式计算，或者使用 print、debug 查看
        x = torch.reshape(x, (-1, 3, 30, 30))
        return x

conv = MyConv()
test_set = torchvision.datasets.CIFAR10(root='../dataset/data',
train=False, transform=torchvision.transforms.ToTensor(),
download=True)
test_loader = DataLoader(test_set, batch_size=64, shuffle=True,
num_workers=0, pin_memory=False, drop_last=False)
writer = SummaryWriter("conv2d")

for i, (imgs, targets) in enumerate(test_loader):
    imgs = conv(imgs)
    # print(imgs.shape)  # torch.Size([64, 6, 30, 30])
    writer.add_images("conv2d", imgs, i)

writer.close()
```

池化层

池化层的理论讲解见Pooling。

最常用的池化层是 `MaxPool2d`（MaxPool2d — PyTorch 2.7 文档 - PyTorch 深度学习库）。此外，如果是只关心输出维度，可以使用自适应池化层。

池化的作用为下采样（当然也有 `UnMaxPool2d` 等上采样的），减小输出维度，用于加快训练、减少计算资源消耗，同时可作为维度对齐之用（比如 Resnet 的残差连接）

示例：

PYTHON

```python
import torch.nn as nn
import torchvision
from torch.utils.data import DataLoader, Dataset
from torch.utils.tensorboard import SummaryWriter

class MyPooling(nn.Module):
    def __init__(self):
        super(MyPooling, self).__init__()
        # stride 默认与 kernel_size 相同，无需额外设置
        self.pool = nn.MaxPool2d(kernel_size=2, padding=0)
    def forward(self, x):
        x = self.pool(x)
        return x

conv = MyPooling()
test_set = torchvision.datasets.CIFAR10(root='../dataset/data',
train=False, transform=torchvision.transforms.ToTensor(),
download=True)
test_loader = DataLoader(test_set, batch_size=64, shuffle=True,
num_workers=0, pin_memory=False, drop_last=False)
writer = SummaryWriter("logs")

for i, (imgs, targets) in enumerate(test_loader):
    imgs = conv(imgs)
    # print(imgs.shape)  # torch.Size([64, 6, 30, 30])
    writer.add_images("pooling", imgs, i)

writer.close()
```

非线性激活层

参考其文档 (torch.nn — PyTorch 2.7 文档 - PyTorch 深度学习库)，接收的输入的第一个维度被视为 batch_size，后面的维度无所谓。

原理和作用都很简单，在此不多说明。

示例：

PYTHON

```python
import torch
import torchvision
from torch import nn
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter

tensor = torch.Tensor([[1, -2],
                       [-0.8, 5]])
# 参考 非线性激活 模块的文档，它们需要输入有一个 batch_size (N)，后面的维度其实无所谓
tensor = torch.reshape(tensor, (1, 1, 2, 2))

test_set = torchvision.datasets.CIFAR10(root='G:\\github\\learnPyTorch\\dataset\\data', train=False, transform=torchvision.transforms.ToTensor(), download=True)
test_loader = DataLoader(test_set, batch_size=64, shuffle=True, num_workers=0, pin_memory=False, drop_last=False)

class MyActivation(nn.Module):
    def __init__(self):
        super(MyActivation, self).__init__()
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, x):
        # output = self.relu(x)
        output = self.tanh(x)
        return output

act = MyActivation()
# output = act(tensor)
# print(f"input: {tensor}")
# print(f"output: {output}")
writer = SummaryWriter("logs")
for i, (imgs, targets) in enumerate(test_loader):
    imgs = act(imgs)
```

```
        writer.add_images("tanh", imgs, i)
    writer.close()
```

归一化层

虽然视频中说用的不多，但是现在用的很多了。 归一化层通过标准化神经网络中间输出的分布，使训练更高效、模型更稳定、性能更好。 对于图像输入，最常用的归一化层是 `BatchNorm2d`，当然在线性层（不是最终输出层）后也会用到 `BatchNorm1d`。 示例：

PYTHON

```python
import torch
from torch import nn

# 准备数据
# 常见于 [N, C, H, W] 维度
input4d = torch.tensor([[1.0, 2.0],
                        [3.0, 4.0]])
input4d = torch.reshape(input4d, (1, 1, 2, 2))
# 常见于 [N, C] 维度
# 训练模式下要求 batch_size (N) 至少为 2
input2d = torch.tensor([[1.0, 3.0, 5.0],
                        [2.0, 4.0, 7.0]])
# 定义网络结构
class BatchNorm(nn.Module):
    def __init__(self):
        super().__init__()
        # 创建这两个类对象时，需要的参数 num_features 都是 通道数
        self.BatchNorm1 = nn.BatchNorm1d(num_features=3)
        self.BatchNorm2 = nn.BatchNorm2d(num_features=1)
    def forward(self, x):
        if len(x.shape) == 4:
            return self.BatchNorm2(x)
        elif len(x.shape) == 2 or len(x.shape) == 3:
            return self.BatchNorm1(x)
        else:
            return None

batchnorm = BatchNorm()
print(f"4d 输入进入 BatchNorm2d 前: {input4d}")
print(f"4d 输入经过 BatchNorm2d 后: {batchnorm(input4d)}")

print(f"2d 输入进入 BatchNorm1d 前: {input2d}")
print(f"2d 输入经过 BatchNorm1d 后: {batchnorm(input2d)}")
```

输出结果：

```
4d 输入进入 BatchNorm2d 前：tensor([[[[1., 2.],
          [3., 4.]]]])
4d 输入经过 BatchNorm2d 后：tensor([[[[-1.3416, -0.4472],
          [ 0.4472,  1.3416]]]], grad_fn=<NativeBatchNormBackward>)
2d 输入进入 BatchNorm1d 前：tensor([[1., 3., 5.],
        [2., 4., 7.]])
2d 输入经过 BatchNorm1d 后：tensor([[-1.0000, -1.0000, -1.0000],
        [ 1.0000,  1.0000,  1.0000]], grad_fn=<NativeBatchNormBackward>)
```

线性层

线性层最常用的就是 `nn.Linear`，要提供输入和输出特征数（神经元个数）作为参数，使用起来还是比较简单的。

示例：

PYTHON

```python
import numpy as np
import torch
import torchvision.datasets
from torch import nn
from torch.utils.data import DataLoader

# 准备数据
test_set = torchvision.datasets.CIFAR10(root='../dataset/data',
train=False, transform=torchvision.transforms.ToTensor(),
download=True)
test_loader = DataLoader(test_set, batch_size=64, shuffle=True,
num_workers=0, pin_memory=False, drop_last=True)

# 定义网络
class Linear(nn.Module):
    def __init__(self, batch_size):
        super(Linear, self).__init__()
        self.linear = nn.Linear(in_features=batch_size * 32 * 32 * 3,
out_features=10)
        self.batch_size = batch_size
    def forward(self, x):
        X = torch.flatten(x)
        x = self.linear(X)
        return x

linear = Linear(batch_size=64)

# 将数据送入网络，获取输出（当然没有训练啦）
for i, (imgs, targets) in enumerate(test_loader):
    imgs = linear(imgs)
    print(np.argmax(imgs.detach().numpy()))
```

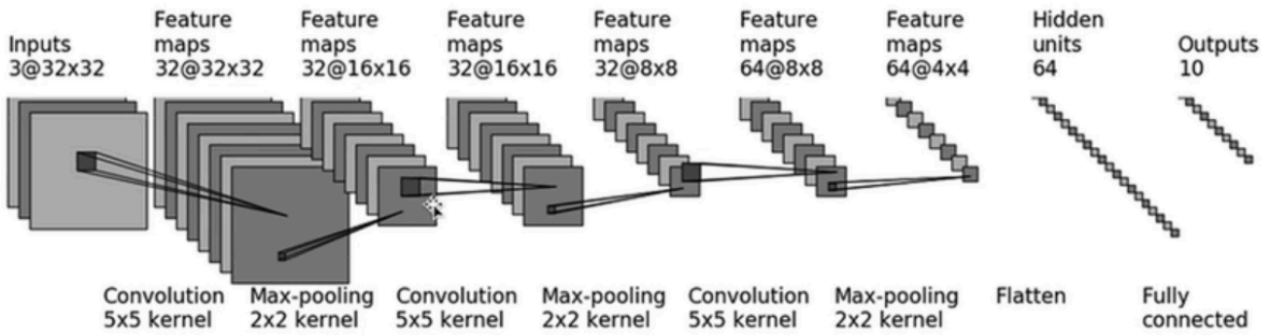
Sequential

`nn.Sequential` 的好处是能将很多层拼接成一个序列，并定义为一个子模型，方便复用，同时避免了过多的变量名。使用起来很简单。

使用 `Sequential` 时还有一个 debug/检查模型结构是否正确的小技巧。注释掉模型的一部分，运行代码，看数据送入会不会报错（主要是维度问题）。用类似二分法的方法，可以快速定位哪

层出错了。

复现 CIFAR10 模型的示例：



PYTHON

```python
import numpy as np
import torchvision
from torch import nn
from torch.nn import Sequential
from torch.utils.data import DataLoader

# 准备数据
test_set = torchvision.datasets.CIFAR10(root='../dataset/data',
train=False, transform=torchvision.transforms.ToTensor(),
download=True)
test_loader = DataLoader(test_set, batch_size=64, shuffle=True,
num_workers=0, pin_memory=False, drop_last=False)

# 定义 CIFAR10 模型
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.model1 = Sequential(
            nn.Conv2d(3, 32, 5, stride=1, padding=2),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(32, 32, 5, stride=1, padding=2),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(32, 64, 5, stride=1, padding=2),
            nn.MaxPool2d(kernel_size=2),
            nn.Flatten(),
            nn.Linear(1024, 10)
        )
    def forward(self, x):
        x = self.model1(x)
        return x

model = Model()
# 预测数据的类别
for i, (imgs, targets) in enumerate(test_loader):
    # imgs: Tensor(64, 3, 32, 32)
    imgs = model(imgs)
    print(imgs.shape)
    print(np.argmax(imgs.detach().numpy())) # 预测结果，虽然没有训练
```

损失函数

损失函数

损失函数的目的是计算模型输出和目标 **(GT)** 之间的差距，同时为更新模型参数提供依据。不同的损失函数有不同的应用场景。

Tips：推荐使用 `torch.tensor` 创建张量，它是一个函数，可以指定类型；`torch.Tensor` 则是一个类，默认创建的类型为 `torch.float32`，不推荐用于初始化。

使用 L1 loss(MAE), L2 loss(MSE), CE(交叉熵损失) 的示例：

PYTHON

```python
import torch
import torch.nn as nn

# 准备数据
output = torch.Tensor([1, 2, 3])
target = torch.Tensor([1, 3, 5])

# 准备损失函数类对象
loss_l1 = nn.L1Loss(reduction='mean')
loss_l2 = nn.MSELoss()
loss_ce = nn.CrossEntropyLoss()

# 计算 l1 和 l2 损失
# 我想要 output/target 的维度为 (N, C, ...)
output = torch.reshape(output, (1, 1, -1))
target = torch.reshape(target, (1, 1, -1))
print(f"l1 loss: {loss_l1(output, target)}")
print(f"l2 loss: {loss_l2(output, target)}")

# 计算 交叉熵损失
# output 的维度应为 (N, C)，现在里面的 3 个数被视为 C 的得分
# target 为类别索引，类别应为 long，而不是 one-hot 向量
output = torch.tensor([[0.1, 0.8, 0.2]])
target = torch.tensor([1], dtype=torch.long)
print(f"cross entropy loss: {loss_ce(output, target)}")
```

神经网络与损失函数结合

将损失函数放到神经网络的训练过程中（当然现在还没训练，因为没到调度器，只是计算了 loss 和 gradient，还没更新参数）。

以Sequential中 CIFAR10 模型为例：

PYTHON

```python
import torchvision
from torch import nn
from torch.nn import Sequential
from torch.utils.data import DataLoader

# 准备数据
test_set = torchvision.datasets.CIFAR10(root='../dataset/data',
train=False, transform=torchvision.transforms.ToTensor(),
download=True)
test_loader = DataLoader(test_set, batch_size=64, shuffle=True,
num_workers=0, pin_memory=False, drop_last=False)

# 定义 CIFAR10 模型
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.model1 = Sequential(
            nn.Conv2d(3, 32, 5, stride=1, padding=2),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(32, 32, 5, stride=1, padding=2),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(32, 64, 5, stride=1, padding=2),
            nn.MaxPool2d(kernel_size=2),
            nn.Flatten(),
            nn.Linear(1024, 10)
        )
    def forward(self, x):
        x = self.model1(x)
        return x

model = Model()
loss_fn = nn.CrossEntropyLoss()
# !: loss 放在这里
for i, (imgs, targets) in enumerate(test_loader):
    # imgs: Tensor(64, 3, 32, 32)
    predictions = model(imgs)
    loss = loss_fn(predictions, targets)
    print(loss)
```

优化器和训练

优化器使用优化算法，根据 `loss` 计算出的梯度，更新模型参数。更新方式根据算法不同而不同。

`loss.backward()` 的执行会计算损失相对于模型参数的梯度，并将这些梯度存储在每个可训练参数的 `.grad` 属性中，`loss` 本身 `grad` 属性为 `None`。

在一次训练开始时，必须要使用 `optimizer.zero_grad()` 清除上一次训练的梯度，否则会导致训练出现问题。

以神经网络与损失函数结合中 CIFAR10 模型为例：

PYTHON

```python
import torch.optim
import torchvision
from torch import nn
from torch.nn import Sequential
from torch.utils.data import DataLoader

# 准备数据
test_set = torchvision.datasets.CIFAR10(root='../dataset/data',
train=False, transform=torchvision.transforms.ToTensor(),
download=True)
test_loader = DataLoader(test_set, batch_size=64, shuffle=True,
num_workers=0, pin_memory=False, drop_last=False)

# 定义 CIFAR10 模型
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.model1 = Sequential(
            nn.Conv2d(3, 32, 5, stride=1, padding=2),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(32, 32, 5, stride=1, padding=2),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(32, 64, 5, stride=1, padding=2),
            nn.MaxPool2d(kernel_size=2),
            nn.Flatten(),
            nn.Linear(1024, 10)
        )
    def forward(self, x):
        x = self.model1(x)
        return x

model = Model()
loss_fn = nn.CrossEntropyLoss()  # 其实是类对象
# 使用优化器，训练模型
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3,
weight_decay=1e-4)
num_epoch = 20
```

```python
for epoch in range(num_epoch):
    print(f"Epoch: {epoch+1}")
    model.train()  # 训练模式
    average_loss = 0.0  # 展示平均损失
    for i, (imgs, targets) in enumerate(test_loader):
        # 清除上一次循环的梯度，一定要写!
        optimizer.zero_grad()
        predictions = model(imgs)
        loss = loss_fn(predictions, targets)
        average_loss += loss.item()
        loss.backward()  # 反向传播算出梯度，存储在对应的可训练参数中
        optimizer.step()  # 优化器更新参数
    average_loss /= len(test_loader)
    print(f"Average Loss: {average_loss}")
```