

Pytorch学习笔记

1.验证安装完成

```
import torch
torch.cuda.is_available()
#有cuda就是True,没有是False
```

2.两个重要函数

1)dir()

```
dir()
dir(torch)
dir(torch.cuda)
#展示包的内容
```

2)help()

```
help()
#查看类如何使用
```

3.Pytorch加载数据

1) Dataset

提供一种方法获取数据和其label

传入: index

输出: Data,label

2)常见的数据组织形式

- 1.多种分类数据分列于多个文件夹, 这些文件夹名字代表数据的标签
- 2.包含训练数据文件夹和标签文件夹, 与训练数据同名的txt文件中含有数据的label
- 3.数据的标签体现在数据的文件名上

3) Dataset的使用

```
from torch.utils.data import Dataset
```

1.Dataset是一个抽象类, 因此数据集类需要继承自该类

```
class MyDataset(Dataset)
    def __init__(self):
        pass
```

2.所有子类应当重写getitem方法, 这个方法是传入index获取其数据和label

```
class MyDataset(Dataset)
    def __getitem__(self, index):
        return data, label
```

3.子类也可以重写len方法，该方法用于提供数据的长度

```
class MyDataset(Dataset)
    def __len__(self):
        return len(self.datadirlist)
```

4. Summery Writer使用

1) SummeryWriter是来自tensorboard模块的可视化工具

使用SummeryWriter时，需要下载tensorboard模块

```
pip install tensorboard -i https://pypi.tuna.tsinghua.edu.cn/simple
```

2) SummeryWriter是一个类，使用时可以传入输出文件夹

```
writer=SummeryWriter("logs")
#将./logs文件夹作为输出文件夹，如果没有该文件夹则创建一个
```

3) SummeryWriter写入单个数据

```
writer.add_scalar(tag:str, scalar_value, global_step:int)
#tag图名，不同图名会生成不同的图
#scalar_value，需要表达的y值
#golbal_step，需要表达的x值，注意是整数
```

4) SummeryWriter写入图片

```
writer.addimage(tag:str, image, global_step:int, dataformats)
#tag图名，不同图名会写入不同的图
#image图像文件流，可以是np.ndarray, PIL.Image, tensor三种类型
#global_step步数，可以拖动步数条显示不同图片
#dataformats, 三种通道的顺序，tensor型不用指定，而ndarray型需要指定"WHC"
```

5. Transforms的使用

1) Transforms是来自pytorchvision包的图片处理工具

```
from torchvision import transforms
```

2) 类型转换工具ToTensor

ToTensor是一个类，其中的方法是非静态的，使用时需要实例化

```
Trans=transforms.ToTensor()
tensorimg=Trans(img)
#img可以支持两种类型，PIL读取的Image类型和cv2读取的np.ndarray类型
```

补充：当类中含有魔术方法call时，如果把实例当成函数使用，则会自动调用魔术方法

3) 图片拉伸工具Resize

Resize实例化时需要传入想要拉伸的大小，类型时tuple(int,int)

```
Reshaper=transforms.Resize(size=(300,500))
#注意这里是传入元组而不是传入两个参数
rspimg=Reshaper(tensorimg)
#魔术方法支持的tensor或PIL类型的图片
#如果没有魔术方法，也可以使用forward方法
rspimg=Reshaper.forward(tensorimg)
```

4)图片归一化工具Normalize

Normalize实例化时需要传入均值和标准差，三通道图片的均值和标准差是长度为3的列表

```
Norm=transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])
image_nor=Norm(tensorimg)
#魔术方法需要填入的参数时tensor类型的图片
#如果没有魔术方法，也可以使用forward方法
image_nor=Norm.forward(tensorimg)
```

6.处理和使用torchvision数据集

1.示例：使用数据集CIFAR0

```
import torchvision
train_set=torchvision.datasets.CIFAR10(root="./dataset",train=True,download=True)
test
_set=torchvision.datasets.CIFAR10(root="./dataset",train=False,download=True)
#root指定数据存放的位置，Train指定该数据集是否是训练数据集，download指定是否下载数据
```

2.使用自定义下载工具下载torchvision数据集

写好数据集名称

```
torchvision.datasets.CIFAR10()
```

使用Ctrl+鼠标左键单击进入数据集详情

```
url = "https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz"
```

找到下载链接，用自带的下载工具进行下载(迅雷，IDM，Motrix等)

放入root指定的对应文件夹

```
train_set=torchvision.datasets.CIFAR10(root="./dataset",train=True,download=True)
)
```

3.处理数据

为数据集传入transforms实例即可

```
from torchvision import transforms
import torchvision
Reshape=transforms.Resize((40,40))
train_set=torchvision.datasets.CIFAR10(root="./dataset",train=True,transforms=Reshape,download=True)
```

通过该参数，为数据集指定Transforms,函数会使用实例并且调用魔术方法进行自动处理

注：这也可能是transforms不使用静态方法的有一种考量，为了方便传入实例

7.使用DataLoader配合SummaryWriter().add_images

1.关于DataLoader

Dataloader是来自torch.data.DataLoader的类，其构造函数含有多个参数。

dataset:Dataset类型，需要加载的类
batch_size:int类型，可选参数，每个图片集的大小
shuffle:bool类型，表示数据是否打乱，默认为False
drop_last:可选参数，默认为False,如果不能整除图片组数，是否要丢弃最后的数据

2.使用DataLoader

DataLoader在建立之后，可以当成一个迭代器使用，是因为其父类定义了相关方法。

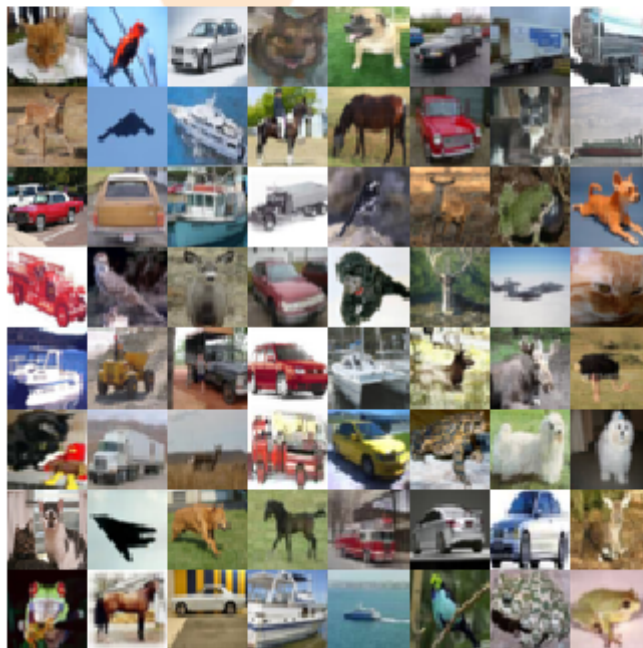
```
Dta=DataLoader(Dataset,batch_size=25,shuffle=True,drop_last=True)
writer=SummaryWriter("logs")
idx=0
for i in Dta:
    img,target=i
    writer.add_images("test",img,global_step=idx)
    idx+=1
```

3.最终效果

test



Step 24



8.继承和使用torch.nn.Module

1.关于torch.nn.Module

torch.nn.Module是nn类的基本骨架，是定义torch神经网络类需要继承的父类。

torch.nn.Module实例化后会存在魔术方法`call`，该方法在调用该实例时会将参数传给forward函数，因此其每一个子类都应该重写forward函数。

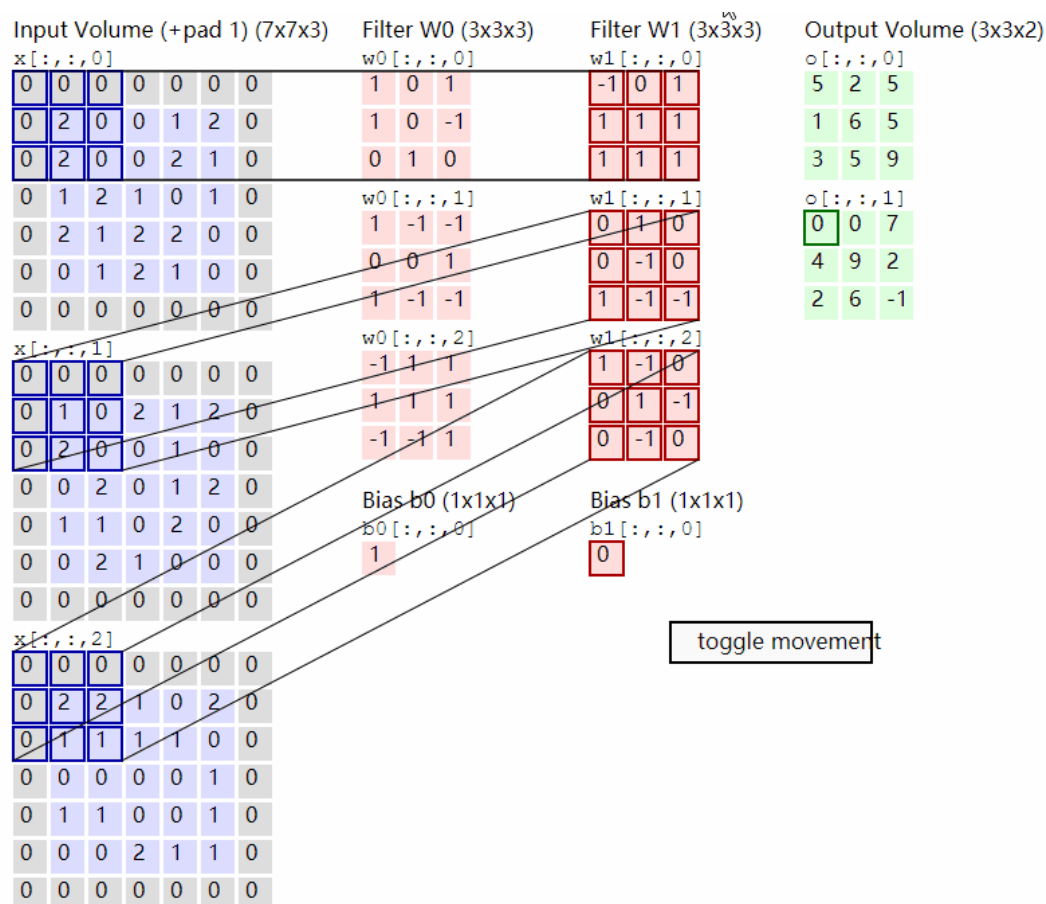
2.使用和构建自定义神经网络

```
from torch.nn import Module
class Mynn(Module):
    def __init__(self):
        super().__init__()
        #调用父类的构造函数
    def forward(self,data):
        data+=1
        return data

customnn=Mynn()
print(customnn(10))
#输出11，调用了父类的魔法函数，使用了子类重写的forward函数
```

9.了解和使用卷积和卷积函数torch.nn.functional.conv2d()

1.卷积的概念



卷积包含两个部分，卷积核和数据。通过使用卷积和对数据的卷积核大小的数据不断做点乘计算，从而生成新的矩阵。

2.使用torch.nn.functional.conv2d

conv族是来自torch.nn.functional类下的方法，包含conv1d,conv2d和conv3d。

使用conv2d时，需要指定tensor类型的卷积核。

```
from torch.nn.functional import conv2d
import numpy as np
from torchvision import transforms

#定义数据和卷积核
#定义转换器
transtools=transforms.ToTensor()
dataset=transtools(np.array([[1,2,9,6,7],[4,3,1,6,9],[7,4,2,6,2],[3,0,9,2,6],
[3,5,7,9,0]]))
ker=transtools(np.array([[1,0,1],[0,1,1],[1,2,2]]))
#ToTensor方法接受一个np.array变量，因此我们需要将列表变量转换为np.array变量

dataset=torch.reshape(dataset,(1,1,5,5))
ker=torch.reshape(ker,(1,1,3,3))
#conv2d接受的数据有要求
output=conv2d(input=dataset,weight=ker,stride=1)
'''input: 输入张量，形状应为 (N, C_in, H_in, W_in)，其中 N 是批次大小，C_in 是输入通道数，H_in 和 W_in 分别是输入的高度和宽度。
weight: 卷积核权重张量，形状为 (out_channels, in_channels/groups, kernel_height, kernel_width)。out_channels 是输出通道数，in_channels/groups 表示每个组的输入通道数。
stride, 卷积步长。'''
```

3.扩展：使用torch.nn.functional.conv2d对灰度图片进行卷积处理

```
import torchvision
import torch
from torchvision import transforms
#读取CIF数据集
dataset=torchvision.datasets.CIFAR10(root="./data",train=False,download=True)
#使用PIL转换为灰度图像
img=transforms.ToTensor()(dataset[0][0].convert('L'))
#使用reshape在不加入信息的情况下扩展一个维度
img=torch.reshape(img,[1]+list(img.shape))
#使用卷积核
ker=[[0.2,0.3,0.5],[0.2,0.6,0.2],[0.6,0.1,0.3]]
#注意卷积核的数据类型需要和图片tensor的数据类型一致，就算double与int,float数据类型卷积都会报错
ker=torch.FloatTensor(ker)
#卷积核维度扩展
ker=torch.reshape(ker,[1,1]+list(ker.shape))
#卷积
output=torch.nn.functional.conv2d(input=img,weight=ker)
#查看卷积结果
imghandled=transforms.ToPILImage()(torch.reshape(output,(1,30,30)))
imghandled.show()
```

10.利用Module骨架构建和使用torch.nn.Conv2d类

1.继承自Module的子类在类对象被当作函数使用时会调用子类的forward方法

```
import torch
class Testclass(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self,i,j,k):
        print(i,j,k)
TC=Testclass()
TC(1,2,3)

#输出1 2 3
```

2.初始化和使用Conv2d类

```
class Myclass(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, image):

        self.imgconv2d=torch.nn.Conv2d(in_channels=3,out_channels=3,kernel_size=3,stride=1,padding=1)
        return self.imgconv2d(image)

    #in_channels表示输入的通道数，out_channels是输出的通道数，kernel_size是卷积核的大小，padding是使用边缘填充，stride是卷积步长
```

3.结合SummaryWriter使用神经网络类

```
import torch
import torchvision
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
class Myclass(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, image):

        self.imgconv2d=torch.nn.Conv2d(in_channels=3,out_channels=3,kernel_size=3,stride=1,padding=1)
        return self.imgconv2d(image)

trans=torchvision.transforms.ToTensor()
Dataset=torchvision.datasets.CIFAR10(root="./data",train=False,download=True,transform=trans)
#由于SummaryWriter需要的是ndarray类或者tensor类，需要使用
torchvision.transforms.ToTensor类进行转换
Dta=DataLoader(Dataset,batch_size=64,shuffle=True,drop_last=True)
writer=SummaryWriter("logs")
idx=0
for i in Dta:
    img,target=i
    img=Myclass()(img)
    #在此处实例化类并且传入数据
    writer.add_images("test",img,global_step=idx)
    idx+=1
writer.close()
```

11.构建和使用torch.nn.MaxPool2d类对二维tensor数据进行池化

1.关于池化

池化和卷积的操作是类似的，池化拥有一个池化核，通过对池化核对齐的数据取最大值，从而实现最大池化效果。

2.MaxPool2d类

MaxPool3d类的初始化

```
#kernel_size:int类型，该参数用于指定池化核的大小。
#stride:int或tuple类型，用于指定在x,y方向池化核步进的步长。注意，该参数的默认值是
kernel_size
#dilation:int类型，指定池化核的空腔大小。例如：当dilation为1且kernel_size=3时，池化核会从
5*5的数据中取出9个点的最大值进行处理。
#ceil_mode:bool类型，表示是否在无法步进到kernel_size^2的数据时忽略这些数据。默认为False,表示
不会忽略。
```

初始化类

```
mypool=torch.nn.MaxPool2d(kernel_size=3)
```

3.MaxPool2d类对象call方法只有一个必要参数

```
img=mypool()(img)
#要求img时tensor对象，维度为4，因此要对3通道RGB图像(3)做ToTensor和reshape变换
```

12.非线性激活类torch.nn.*

1.非线性激活类

非线性激活类有很多种，根据不同的论文研发。

例如：

```
torch.nn.ReLU()
torch.nn.Sigmoid()
```

这些类在初始化时不需要指定参数，在使用时会把tensor数据类型的每一个数据应用变换。

2.使用方法

```
from torch import ReLU
class Linearclass(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def dataLinear(self,data):
        Re=ReLU()
        return Re(data)
```

该层的实质是对数据应用一个函数。

13.矩阵展开flatten和线性变换类Linear

1.矩阵展开方法flatten

flatten是能够将高维矩阵按顺序展开成一维列表的方法，其来源是torch，必要参数只有一个，必须是tensor类型的矩阵。

```
import torch
data=torch.Tensor([[1,2,3],[4,5,6],[7,8,9]])
output=torch.flatten(data)
print(output)
#tensor([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

2.实现类似功能的reshape方法

reshape方法也是torch下的矩阵变化方法，当某维度的值为阶数写-1时，会自动推导该维度的阶数。

```
import torch
data=torch.Tensor([[1,2,3],[4,5,6],[7,8,9]])
output=torch.reshape(data,(1,1,-1))
print(output)
#tensor([[[[1., 2., 3., 4., 5., 6., 7., 8., 9.]])])
```

很明显，reshape方法不会改变数据的维数。

3.线性变换类Linear

Linear类是一个和卷积类池化类类似的，初始化的必须参数只有数据大小和需求变换大小的类。

```
import torch
from torch import Linear
class Linearclass(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def datalinear(self,data):
        Lin=Linear(64,10)
        return Lin(data)
```

通过这种方法，64长度的tensor数据变为了10长度的。

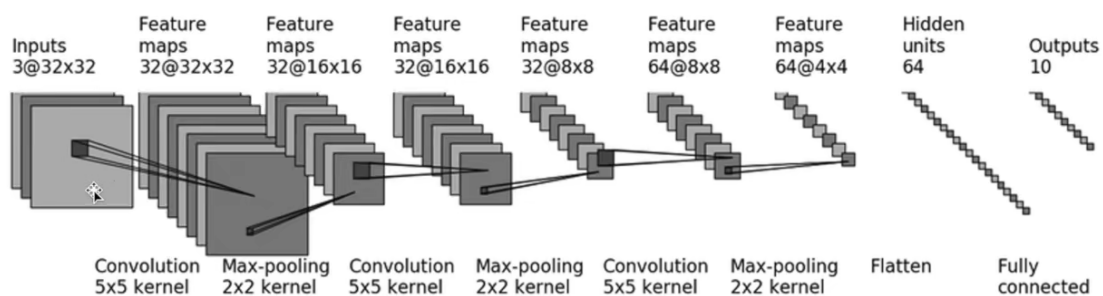
14.使用Sequential类构建神经网络

1.Sequential类是一个可以串联类对象的torch.nn类

Sequential的参数是类的对象，要求该对象继承torch.nn.Module骨架，而且完成了forward函数的重写。

2.Sequential类的使用

例如这张图：



CSDN @qq_932016561

使用Sequential处理构建层

```
self.prenet=Sequential(
    Conv2d(in_channels=3,out_channels=32,kernel_size=(5,5),padding=2),
    MaxPool2d(kernel_size=2),
    Conv2d(in_channels=32,out_channels=32,kernel_size=(5,5),padding=2),
    MaxPool2d(kernel_size=2),
    Conv2d(in_channels=32,out_channels=64,kernel_size=(5,5),padding=2),
    MaxPool2d(kernel_size=2),
    Flatten(),
    Linear(1024,64),
    Linear(64,10)
)
```

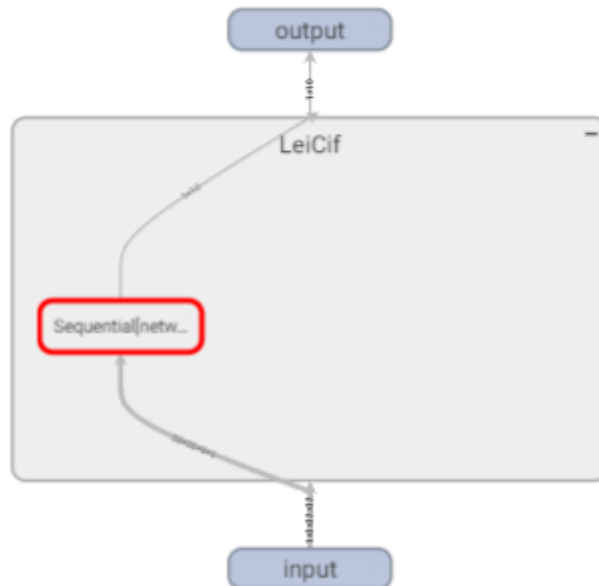
相当于将所有层整合到了Sequential类中。

3.SummaryWriter类结合神经网络Sequential类的使用

SummaryWriter类能够和Sequential类结合使用，在tensorboard中打印出模型的具体层数和流程。

```
writer=SummaryWriter("logs")
writer.add_graph(netModule,input)
```

效果如下图



4.Sequential类调试小技巧

编写类ptn，能够打印Sequential计算过程中的数据尺寸，从而实现调试。

```
class prn(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self,data):
        print(data.shape)
        return data
```

15 损失函数的使用

1.注意损失函数的输入输出数据类型

不同的损失函数有着不同的输入输出数据类型。

CrossEntropyLoss类

```
#CrossEntropyLoss类是一个损失函数类，其初始化时没有必要参数，但是其使用时需要传入各个分类的计算概率和标签。
#该类是一个专用于衡量分类效果的类
CELoss=CrossEntryLoss()
PerLoss=CELoss(Modle(img),target)
#该类的计算原理在文档中指定了，能够使得当标签的概率越高，非标签的概率越低时，损失函数的大小越小
```

L1Loss类

```
#CrossEntropyLoss类是一个损失函数类，其初始化时没有必要参数，使用时需要传入结果tensor类和标签tensor类
#该类会计算结果和标签的绝对值差和
#其初始化时有一个可选参数reduction，默认为"mean"，表示求出绝对值差和后会对样本大小求平均
#当reduction取"sum"时，只求和
LossFunc=L1Loss()
PerLoss=LossFunc(torch.Tensor([1,2,3]),torch.Tensor([2,1.1,3.3]))
#如果为reduction="sum":PerLoss=2.2
#如果为reduction="mean":PerLoss=0.73
```

MSELoss类

```
#CrossEntropyLoss类是一个损失函数类，其初始化时没有必要参数，使用时需要传入结果tensor类和标签tensor类
#该类会计算结果和标签的绝对值差平方和
LossFunc=MSELoss()
PerLoss=LossFunc(torch.Tensor([1,2,3]),torch.Tensor([2,1.1,3.3]))
#PerLoss=0.63
```

2.反向传播和自动求导机制

损失函数类的backward方法能够对每层神经网络求取梯度，从而为优化器更新提供依据。

```
LossFunc.backward()
```

使用该方法后，torch会对每层网络自动求梯度，并且将梯度累加到Sequential类的Module实例中各层的权重和偏置类的grad属性下，从而使得优化器能够通过step()方法取得这些属性。

16.optimizer的定义和使用

1.torch提供的部分optimizer

```
#SGD (Stochastic Gradient Descent): 最基础的随机梯度下降法。
torch.optim.SGD()
#Adam (Adaptive Moment Estimation): 结合了动量和自适应学习率的方法，是目前非常流行的优化器。
torch.optim.Adam()
#RMSprop (Root Mean Square Propagation): 使用平方梯度的指数移动平均来调整学习率。
torch.optim.RMSprop()
```

2.optimizer需要用torch.nn.Module.parameters()和学习率lr来初始化

```
optim=Adam(MyNet.parameters(),lr=0.01)
#lr指的是学习率,learning rate
```

3.optimizer的使用和运作过程

optimizer配合损失函数使用，损失函数LossFunc将自动求导结果在Backward()调用时写入每一层模型的权重和偏置类的grad参数下，当optimizer的step()方法调用时，会获取这些grad梯度以此为依据对权重调整。

由于LossFunc.backward()方法是将梯度累加到网络中，要将这些梯度在每次清零。

17.使用和修改torchvision提供的模型

1.torchvision的模型位于torchvision.models中

比如使用torchvision提供的视觉分类模型vgg19:

```
import torchvision
vgg19=torchvision.models.vgg19(weight="IMAGENET1K_V1",progress=True)
#这两个参数是官方文档给定的，weight表示下载的数据是否使用与训练好的模型，progress表示是否显示
#下载进度
```

2.为模型加入层

这是表示VGG模型的网络结构。

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
      ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
      ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
      ceil_mode=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace=True)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
      ceil_mode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace=True)
```

```

        (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
            ceil_mode=False)
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
    (classifier): Sequential(
      (0): Linear(in_features=25088, out_features=4096, bias=True)
      (1): ReLU(inplace=True)
      (2): Dropout(p=0.5, inplace=False)
      (3): Linear(in_features=4096, out_features=4096, bias=True)
      (4): ReLU(inplace=True)
      (5): Dropout(p=0.5, inplace=False)
      (6): Linear(in_features=4096, out_features=1000, bias=True)
    )
  )
)

```

向vgg加入一个线形层，名字为linear

```
vgg19.add_module(name="linear", Linear(in_features=1000, out_features=10))
```

向vgg.classifier加入一个线性层，名字为7

```
vgg19.classifier.add_module(name="7", Linear(in_features=1000, out_features=10))
```

3. 替换模型层

虽然向模型加入层很方便，但是有时候新加入的层不一定被模型传播，加入的层就变为没用的层了。

这时我们需要替换模型中有用的层。

比如以图像识别网络ResNet50，其最后结构是这样的：

```

(avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
(fc): Linear(in_features=2048, out_features=1000, bias=True)

```

当我们向model.resnet其中加入线形层Refc，模型是这样的：

```

(avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
(fc): Linear(in_features=2048, out_features=1000, bias=True)
(refc): Linear(in_features=1000, out_features=100, bias=True)

```

但是输出的时候outputs仍然是size(1,1000),说明模型没有经过该层。

那么我们可以使用：

```

model.fc=Sequential(Linear(in_features=2048,out_features=1000,bias=True),
                    Lineat(in_features=1000,out_features=100,bias=True))

```

16.完整的模型训练套路

1.构建Dataset

Dataset可以使用torchvision提供的数据集

```

Dataset=torchvision.dataset.CIFAR10(root="./data",download=True,transfer=torchvi
son.transforms.ToTensor())

```

也可以自己写Dataset数据集，使用散装的图片文件等，那么要重写`getitem`类和可选择重写`len`类

```
datadir = "./data/train_data_all"

class saku_dataset(Dataset):
    def __init__(self, path, mode="train"):
        super().__init__()
        self.Dire = open("./data/targetsdir/targets.txt", "r")
        self.Dire = json.load(self.Dire)
        self.filedir = path
        self.path = datadir + "/" + path + "/" + mode

    def __getitem__(self, idx):
        img = Image.open(self.path + "/" + os.listdir(self.path)[idx])
        if img.mode == "RGBA":
            img = img.convert("RGB")
        img = torchvision.transforms.ToTensor()(img)
        target = int(self.Dire[self.filedir])
        return img, target

    def __len__(self):
        return len(os.listdir(self.path))
```

2.构建DataLoader

`dataloader`是一个数据迭代器类，要求用一个`Dataset`对象来初始化，有许多可选参数。

```
datasets_test = DataLoader(dataset=test_data, batch_size=1, shuffle=True,
                             drop_last=True)
```

3.定义模型类

模型类必须是继承自`nn.Module`的类，为了保证其功能，要将层添加到正向传播方法`forward`中。

```
class MyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.mod=Sequential(
            ...
        )
    def forward(self, img):
        return self.mod(img)
```

4.定义损失函数

损失函数使用`pytorch`提供的损失函数即可。

```
LossFunc=torch.nn.CrossEntropyLoss()
```

在`pytorch`官方文档选择合适的损失函数。

5.定义优化器

在pytorch官方文档中找到合适的优化器，比如亚当Adam和梯度下降SGD等。

```
optim=torch.optim.Adam(model_rewriter.parameters())
```

亚当的必要参数只有模型类对象的parameters参数迭代器。学习率会自适应使用。

6.构建训练和测试流程

我们从训练和测试的迭代器中提取数据和标签，用于训练和测试数据。

训练数据：

```
for data in dataset_train:
    img,target=data
    #取出数据
    output=model(img)
    #模型处理数据
    optim.zero_grad()
    #清空梯度
    PerLoss=LossFunc(output,target)
    #计算损失
    PerLoss.backward()
    #反向传播叠加梯度
    optim.step()
    #步进优化器
```

测试数据：

```
Full_Loss=0
for data in dataset_test:
    with torch.no_grad():
        img,target=data
        #取出数据
        output=model(img)
        #模型处理数据
        PerLoss=LossFunc(output,target)
        #计算损失
        Full_Loss+=PerLoss
```

这是一个训练轮次的代码，我们需要训练更多轮次从而找到模型的最好表现位置。

在测试数据时可以使用SummeryWriter的add_scalar方法绘制训练损失图，从而达到最好效果。

```
writer.add_scalar("testloss",scalar_value=Full_Loss,global_step=epoch)
```

将轮数写入步数中，从而能够很好展示随着训练进程损失的变化。