1. **模型定义**

import torch.nn as nn

import torch.nn.functional as F

import torch.nn.init as init

class MLP\_cls(nn.Module):

def \_\_init\_\_(self,in\_dim=3\*32\*32):

super(MLP\_cls,self).\_\_init\_\_()

self.lin1 = nn.Linear(in\_dim,128)

self.lin2 = nn.Linear(128,64)

self.lin3 = nn.Linear(64,10)

self.relu = nn.ReLU()

init.xavier\_uniform\_(self.lin1.weight)

init.xavier\_uniform\_(self.lin2.weight)

init.xavier\_uniform\_(self.lin3.weight)

def forward(self,x):

x = x.view(-1,3\*32\*32)

x = self.lin1(x)

x = self.relu(x)

x = self.lin2(x)

x = self.relu(x)

x = self.lin3(x)

x = self.relu(x)

return x

代码说明：

1. class MLP\_cls(nn.Module):：定义了一个名为MLP\_cls的类，它继承自nn.Module，这是PyTorch中构建神经网络模型的基类。
2. def \_\_init\_\_(self,in\_dim=3\*32\*32):：这是类的构造函数，用于初始化模型的各个组件。in\_dim参数默认值为图像的大小（3通道，每个通道32x32像素），它指定了输入数据的维度。
3. 在构造函数中，首先调用了父类nn.Module的构造函数super(MLP\_cls,self).\_\_init\_\_()。
4. 接着，通过self.lin1 = nn.Linear(in\_dim,128)定义了一个线性变换层（全连接层），输入维度为in\_dim，输出维度为128。这一层被赋予了名为lin1的变量。
5. 同样地，定义了两个额外的全连接层，分别为self.lin2 = nn.Linear(128,64)和self.lin3 = nn.Linear(64,10)。这两个层的输出维度分别是64和10。
6. 使用self.relu = nn.ReLU()创建了一个ReLU激活函数的实例，以便在模型的前向传播过程中使用。
7. 使用init.xavier\_uniform\_(self.lin1.weight)、init.xavier\_uniform\_(self.lin2.weight)和init.xavier\_uniform\_(self.lin3.weight)对模型的权重进行了初始化。这里使用的是Xavier初始化方法，它可以帮助加速收敛。
8. def forward(self,x):：定义了模型的前向传播过程。输入参数x是模型的输入数据。
9. x = x.view(-1,3\*32\*32)：首先对输入数据x进行了reshape操作，将其转换为二维张量。-1表示该维度的大小由数据的总大小和其他维度的大小来自动推断，3\*32\*32表示一个3通道、32x32像素的图像被展平成一维向量。
10. x = self.lin1(x)：将输入数据通过第一个全连接层lin1进行线性变换。
11. x = self.relu(x)：将经过第一个全连接层的结果通过ReLU激活函数进行非线性变换。
12. 类似地，依次通过第二个全连接层lin2和第三个全连接层lin3，并在每一层之后应用ReLU激活函数。
13. 最后，将最终的输出返回。在这个示例中，没有应用softmax函数，因为通常在模型的损失函数中集成了softmax操作。

这个模型的结构由三个全连接层组成，每个全连接层后面跟着一个ReLU激活函数。

1. **训练过程**

import torch

import torch.nn as nn

import torchvision

from torch.utils.data import DataLoader

import torch.optim as optim

seed = 42

torch.manual\_seed(seed)

batch\_size\_train = 64

batch\_size\_test = 64

epochs = 10

learning\_rate = 0.01

momentum = 0.5

net = MLP\_cls()

train\_loader = torch.utils.data.DataLoader(

torchvision.datasets.CIFAR10('./data/', train=True, download=True,

transform=torchvision.transforms.Compose([

torchvision.transforms.ToTensor(),

torchvision.transforms.Normalize(

(0.5,), (0.5,))

])),

batch\_size=batch\_size\_train, shuffle=True)

test\_loader = torch.utils.data.DataLoader(

torchvision.datasets.CIFAR10('./data/', train=False, download=True,

transform=torchvision.transforms.Compose([

torchvision.transforms.ToTensor(),

torchvision.transforms.Normalize(

(0.5,), (0.5,))

])),

batch\_size=batch\_size\_test, shuffle=True)

optimizer = optim.SGD(net.parameters(), lr=learning\_rate,momentum=momentum)

criterion = nn.CrossEntropyLoss()

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Begin Training\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

net.train()

for epoch in range(epochs):

run\_loss = 0

correct\_num = 0

for batch\_idx, (data, target) in enumerate(train\_loader):

out = net(data)

\_,pred = torch.max(out,dim=1)

optimizer.zero\_grad()

loss = criterion(out,target)

loss.backward()

run\_loss += loss

optimizer.step()

correct\_num += torch.sum(pred==target)

print('epoch',epoch,'loss {:.2f}'.format(run\_loss.item()/len(train\_loader)),'accuracy {:.2f}'.format(correct\_num.item()/(len(train\_loader)\*batch\_size\_train)))

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Begin Testing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

net.eval()

test\_loss = 0

test\_correct\_num = 0

for batch\_idx, (data, target) in enumerate(test\_loader):

out = net(data)

\_,pred = torch.max(out,dim=1)

test\_loss += criterion(out,target)

test\_correct\_num += torch.sum(pred==target)

print('loss {:.2f}'.format(test\_loss.item()/len(test\_loader)),'accuracy {:.2f}'.format(test\_correct\_num.item()/(len(test\_loader)\*batch\_size\_test)))

代码说明：

1. 首先导入了必要的PyTorch模块以及相关的库。
2. 使用torch.manual\_seed(seed)设置了随机种子，以确保结果的可重复性。
3. 定义了训练和测试阶段的批次大小（batch\_size\_train和batch\_size\_test）、训练周期数（epochs）、学习率（learning\_rate）和动量（momentum）等超参数。
4. 创建了一个MLP分类器的实例net = MLP\_cls()，这是在前面定义的MLP\_cls类的基础上实例化的一个对象。
5. 使用torchvision.datasets.CIFAR10加载了CIFAR-10数据集，并通过torch.utils.data.DataLoader创建了训练和测试数据集的数据加载器（train\_loader和test\_loader），其中对数据进行了标准化处理。
6. 使用随机梯度下降（SGD）优化器optim.SGD，将MLP模型的参数传递给优化器。
7. 定义了交叉熵损失函数nn.CrossEntropyLoss()。
8. 进入训练阶段，通过net.train()将模型设置为训练模式。然后进行多个周期的训练循环，每个周期中，通过遍历训练数据加载器中的每个批次来进行训练。
9. 在每个批次中，将数据传递给模型进行前向传播，计算输出，计算损失，然后反向传播并更新模型参数。同时，计算损失和正确分类的数量。
10. 打印每个周期的平均损失和准确率。
11. 进入测试阶段，通过net.eval()将模型设置为评估模式。然后遍历测试数据加载器中的每个批次，对测试数据进行预测，计算损失和正确分类的数量。
12. 打印测试集上的平均损失和准确率。

这段代码实现了一个简单的MLP模型在CIFAR-10数据集上的训练和测试过程。