**实验报告**

姓名： 林炬乙 学号： 3180103721

课程名称： 数字图像处理 任课老师： 项志宇

实验名称： cifar-10物品分类识别 实验日期： 2021/6/23

# 1 实验目的和要求

（分点简要说明本次实验需要进行的工作和最终的目的）

实现一个CNN，进行CIFAR-10物品分类识别。

# 2 实验原理

PyTorch 是一个建立在 Torch 库之上的 Python 包，旨在加速深度学习应用。PyTorch 提供一种类似 NumPy 的抽象方法来表征张量（或多维数组），它可以利用 GPU 来加速训练。PyTorch 的关键数据结构是张量，即多维数组。其功能与 NumPy 的 ndarray 对象类似，我们可以使用 torch.Tensor() 创建张量。

CIFAR-10是一个接近普适物体的彩色图像数据集。CIFAR-10 是由Hinton 的学生Alex Krizhevsky 和Ilya Sutskever 整理的一个用于识别普适物体的小型数据集。一共包含10 个类别的RGB 彩色图片：飞机（ airplane ）、汽车（ automobile ）、鸟类（ bird ）、猫（ cat ）、鹿（ deer ）、狗（ dog ）、蛙类（ frog ）、马（ horse ）、船（ ship ）和卡车（ truck ）.

每个图片的尺寸为32 × 32 ，每个类别有6000个图像，数据集中一共有50000 张训练图片和10000 张测试图片。CIFAR-10是32\*32的彩色图， 直接读入官网的数据就是处理好的dict格式.如果没有, 可以利用numpy的 frombuffer 读入,可能还需要reshape处理. 相比于手写字符， CIFAR-10 含有的是现实世界中真实的物体，不仅噪声很大，而且物体的比例、特征都不尽相同，这为识别带来很大困难。直接的线性模型如Softmax 在CIFAR-10 上表现得很差。

## 梯度下降

以图像分类为例，以代表图像 RGB 值的数组为输入，生成一组分类得分的输出。通过训练神经网络，我们是在最小化一个损失函数。这个损失函数的值衡量了我们网络的性能在给定数据集上离完美还差多少。一开始如果随机初始化了权值，所以神经网络可能会把猫的图片归类为人。这样的情况对应的是损失函数较高点，此处的网络性能十分差，因此损失也很高。首先要做的就是查看 x-y 平面中所有可能的方向，看看哪个方向是损失函数的值下降最陡峭的方向。这个就是我们必须移动的方向，它恰恰与梯度的方向相反。梯度是高维导数的另一种说法，它给出了最陡峭的上升方向。我们沿着梯度的方向进行下降，所以就叫做梯度下降。

# 3 实验内容

## 3.1 加载和初始化训练和测试的数据集

自己定义数据集, 转换、归一化并载入，我们将它们转换为归一化范围[-1，1]的张量, 展示一些训练图像，展示的同时会显示物品类别.

下面的代码从文件读入batch,其实只有一个训练集文件

def unpickle(file):  
 import pickle  
 with open(file, 'rb') as fo:  
 dict = pickle.load(fo, encoding='bytes') # dict有四个key，其中data是10000\*6072的图片数据，labels  
 return dict # 是标成0-9分类，还有其他key有字符串格式的分类说明  
  
  
def load\_data(folder, data\_name):  
 data\_batch1 = unpickle(folder + data\_name)  
 train\_set\_x = data\_batch1[b'data']  
 train\_set\_x = train\_set\_x.reshape(10000, 3, 32, 32) # 转成矩阵格式  
 train\_set\_x = np.transpose(train\_set\_x, (0, 2, 3, 1)) # 改成 10000 \* 32 \* 32\* 3  
 binary\_set\_x = train\_set\_x / 255 #归一化  
 train\_set\_y = data\_batch1[b'labels']  
 train\_set\_y = np.array(train\_set\_y) #载入labels  
 return binary\_set\_x, train\_set\_y

自己定义了数据集Dataset结构

class DealDataset(Dataset):  
  
 def \_\_init\_\_(self, folder, data\_name, transform=None):  
 (train\_set, train\_labels) = load\_data(folder, data\_name) # 其实也可以直接使用torch.load(),读取之后的结果为torch.Tensor形式  
 self.train\_set = train\_set  
 self.train\_labels = train\_labels  
 self.transform = transform  
  
 def \_\_getitem\_\_(self, index):  
 img, target = self.train\_set[index], int(self.train\_labels[index])  
 if self.transform is not None:  
 img = self.transform(img)  
 return img, target  
  
 def \_\_len\_\_(self):  
 return len(self.train\_set)

转换、归一化并载入，我们将它们转换为归一化范围[-1，1]的张量：

transforms.Compose()类。这个类的主要作用是串联多个图片变换的操作。

transforms.Compose([transforms.ToTensor(),transforms.Normalize(std=(0.5,0.5,0.5),mean=(0.5,0.5,0.5))])，则其作用就是先将输入归一化到(0,1)，再使用公式"(x-mean)/std"，将每个元素分布到(-1,1)

ToTensor()将shape为(H, W, C)的nump.ndarray或img转为shape为(C, H, W)的tensor

transform = transforms.Compose(  
 [transforms.ToTensor(),  
 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])  
  
train\_dataset = DealDataset(r'./cifar/', "data\_batch\_1", transform=transform)  
test\_dataset = DealDataset(r'./cifar/', "test\_batch", transform=transform)

PyTorch 具备数据加载器和处理器，可用于不同的数据集。数据集下载好后，你可以随时使用。你还可以将数据包装进 PyTorch 张量，创建自己的数据加载器类别。pytorch 的数据加载到模型的操作顺序是这样的：

① 创建一个 Dataset 对象  
② 创建一个 DataLoader 对象  
③ 循环这个 DataLoader 对象，将img, label加载到模型中进行训练

train\_loader = torch.utils.data.DataLoader(train\_dataset, batch\_size=BATCH\_SIZE, shuffle=True)  
test\_loader = torch.utils.data.DataLoader(train\_dataset, batch\_size=BATCH\_SIZE, shuffle=False)  
  
classes = ('plane', 'car', 'bird', 'cat',  
 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

展示一些训练图像，展示的同时会显示物品类别：

# 图片可视化  
images, labels = next(iter(train\_loader)) #使用iter(dataloader)返回的是一个迭代器，然后可以使用next访问；  
img = torchvision.utils.make\_grid(images)  
img = img / 2 + 0.5  
img = img.numpy().transpose(1, 2, 0)  
print(' '.join('%5s' % classes[labels[j]] for j in range(8)))  
plt.imshow(img)  
plt.show()



## 2. 定义卷积神经网络和损失函数

对网络进行一定的修改，以获取三通道图像。

继承 nn.Module 的模块主要重载 init、 forward、 和 extra\_repr 函数

Modules也可以包括其他Ｍodules，运行使用树结构来嵌入，可以将子模块给模型赋予属性，从下列看出，self.conv1 , self.conv2等是模型的子模型

class Net(nn.Module):  
 def \_\_init\_\_(self):  
 super(Net, self).\_\_init\_\_()  
 self.conv1 = nn.Conv2d(3, 6, 5)  
 self.pool = nn.MaxPool2d(2, 2)  
 self.conv2 = nn.Conv2d(6, 16, 5)  
 self.fc1 = nn.Linear(16 \* 5 \* 5, 120)  
 self.fc2 = nn.Linear(120, 84)  
 self.fc3 = nn.Linear(84, 10)  
  
 def forward(self, x):  
 x = self.pool(F.relu(self.conv1(x)))  
 x = self.pool(F.relu(self.conv2(x)))  
 x = x.view(-1, 16 \* 5 \* 5)  
 x = F.relu(self.fc1(x))  
 x = F.relu(self.fc2(x))  
 x = self.fc3(x)  
 return x  
  
net = Net()  
net = net.double()  
criterion = nn.CrossEntropyLoss()  
optimizer = optim.SGD(net.parameters(), lr=0.002, momentum=0.9) #优化器

继承 nn.Module 的神经网络模块在实现自己的 \_\_init\_\_ 函数时，一定要先调用 super().\_\_init\_\_()。只有这样才能正确地初始化自定义的神经网络模块，否则会缺少上面代码中的成员变量而导致模块被调用时出错。实际上，如果没有提前调用 super().\_\_init\_\_()，在增加模块的 parameter 或者 buffer 的时候，被调用的 \_\_setattr\_\_ 函数也会检查出父类 nn.Module 没被正确地初始化并报错。

激活函数用relu，当x>0时，relu 的导数是 1。当 x <=0 时，relu 的导数是 0。而梯度下降算法其实就是链式求导，那么必然会导致多个梯度连乘，而导数为 1 的话，可以保证激活函数对其他的信息不会进行缩放。并且 1 \* 1 还是 1。如果值为1，梯度保持值不变进行前向传播；如果值为0 ,梯度从该位置停止前向传播。

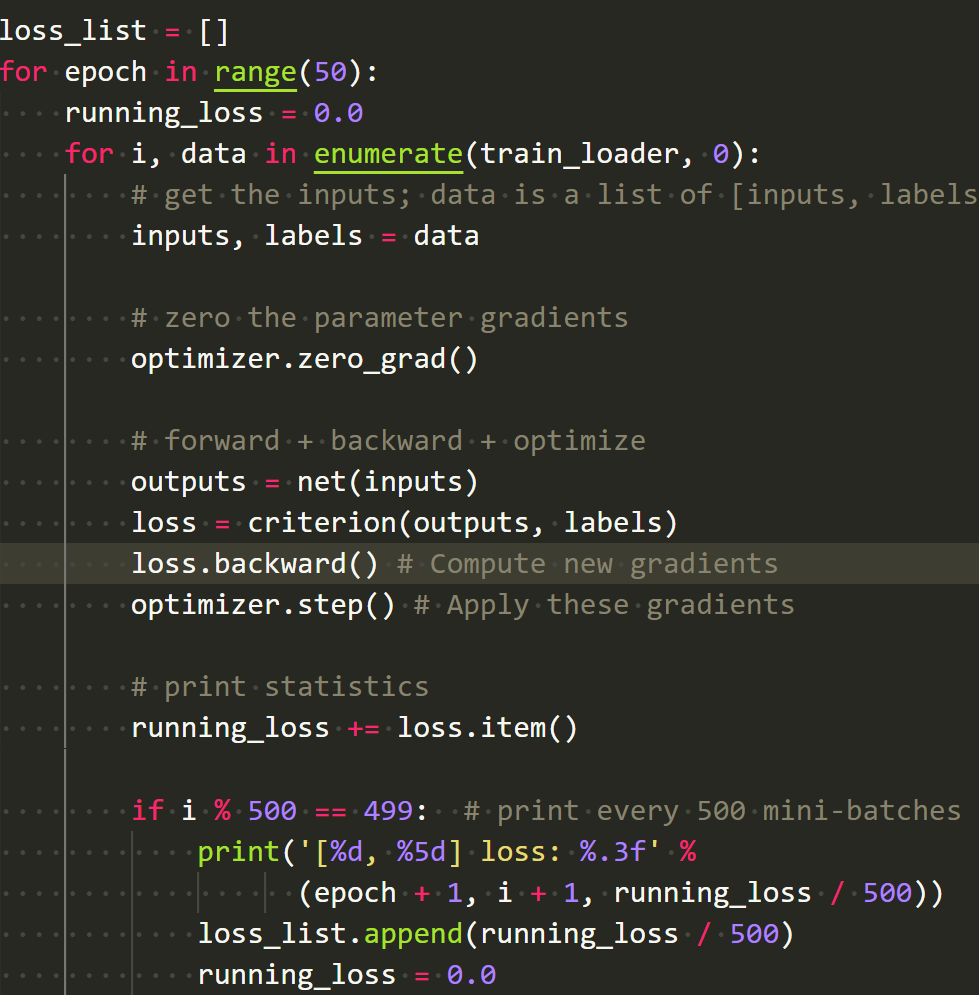
全连接神经网络是一种最基本的神经网络结构，英文为Full Connection，所以一般简称FC。FC的准则很简单：神经网络中除输入层之外的每个节点都和上一层的所有节点有连接。我们还需要使用优化函数（SGD），并运行反向传播。

我们采用交叉熵函数作为损失函数, 交叉分类问题中，交叉熵函数是比较常用也是比较基础的损失函数，能够表征真实样本标签和预测概率之间的差值.

## 3. 根据训练数据训练网络

在实际运行中，我发现物品分类需要比较多的训练次数才有比较好的效果，这里取了epoch=50。在训练结束后绘制loss曲线。

PyTorch提供两种求梯度的方法：backward() and torch.autograd.grad() ，他们的区别在于前者是给叶子节点填充.grad字段，而后者是直接返回梯度.



## 4. 在测试数据上测试网络

我们先试试测试一个batch（这里batch=8）的数据，并且把它显示出来看看效果。选一个batch的测试数据，输出它的对应类别并显示出来，然后用我们的网络测试它，输出分类结果并对比。

# 测试  
images, labels = next(iter(test\_loader))  
img = torchvision.utils.make\_grid(images)  
img = img / 2 + 0.5  
img = img.numpy().transpose(1, 2, 0)  
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(8)))  
plt.imshow(img)  
plt.show()  
  
net = Net().double()  
net.load\_state\_dict(torch.load(PATH))  
  
outputs = net(images)  
  
\_, predicted = torch.max(outputs, 1)  
  
print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]  
 for j in range(8)))

然后测试整个测试数据集的结果并输出，这里我输出了整个数据集的测试准确率以及每个类别的准确率：

torch.no\_grad() 会关闭自动求导引擎，因此能节省显存和加速。

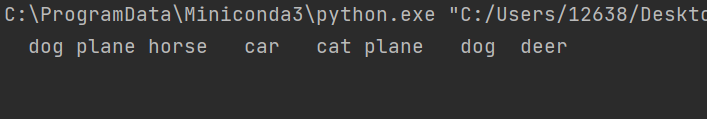
correct = 0  
total = 0  
with torch.no\_grad():  
 for data in test\_loader:  
 images, labels = data  
 outputs = net(images)  
 \_, predicted = torch.max(outputs.data, 1)  
 total += labels.size(0)  
 correct += (predicted == labels).sum().item()  
  
print('Accuracy of the network on the 10000 test images: %d %%' % (  
 100 \* correct / total))  
  
class\_correct = list(0. for i in range(10))  
class\_total = list(0. for i in range(10))  
with torch.no\_grad():  
 for data in test\_loader:  
 images, labels = data  
 outputs = net(images)  
 \_, predicted = torch.max(outputs, 1)  
 c = (predicted == labels).squeeze()  
 for i in range(4):  
 label = labels[i]  
 class\_correct[label] += c[i].item()  
 class\_total[label] += 1  
  
  
for i in range(10):  
 print('Accuracy of %5s : %2d %%' % (  
 classes[i], 100 \* class\_correct[i] / class\_total[i]))

# 4 实验结果和分析

（使用图片和文字叙述实验结果，并对这些结果进行适当分析）

载入数据时，可视化一部分图片：

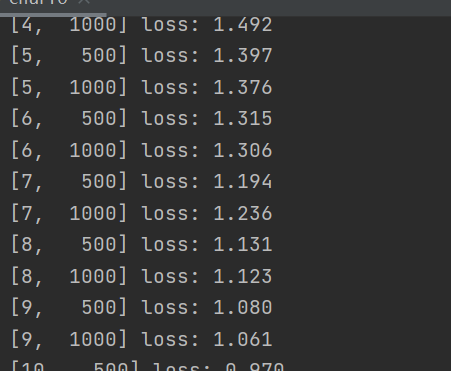
输出类别：



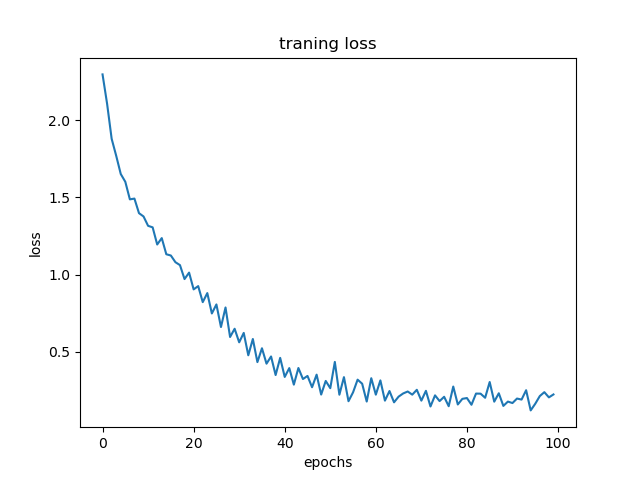
显示图片，仔细观察可以和类别对上：



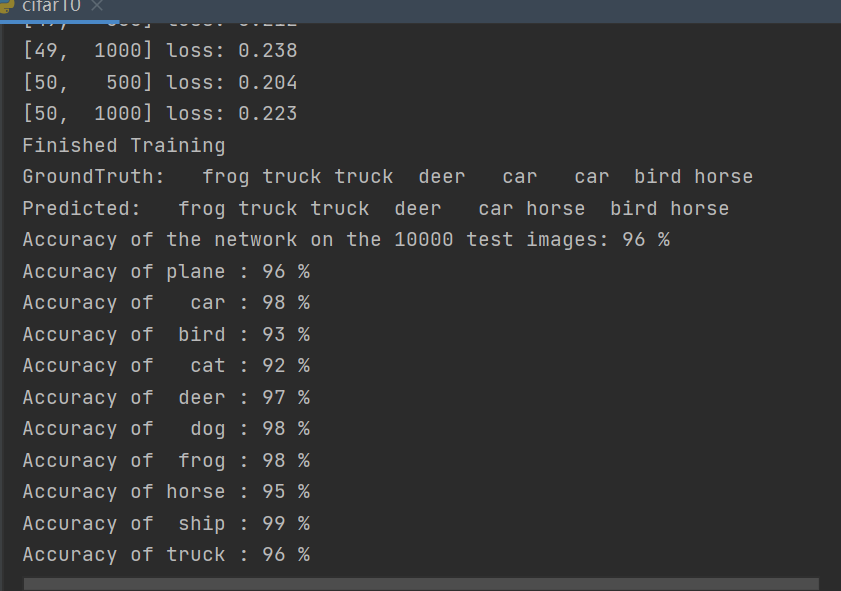
训练过程截图：



loss的变化曲线：



测试结果输出如下，其中第一行是测试样本对应分类，第二行是我们的模型对测试样本的分类。第三行 是我们的模型对10000个测试图片的测试准确率，并且给出了所有类别的测试准确率。

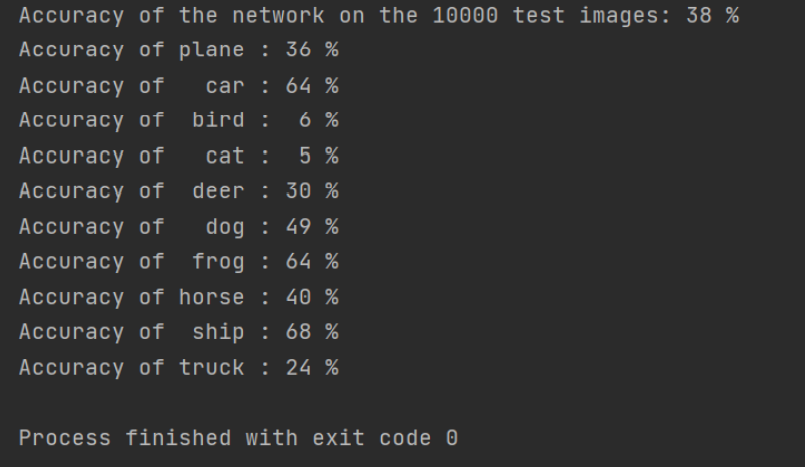


可以看到我们达到了96%的准确率，其中轮船识别率最高，猫的识别率最低。

编程体会:

搭建网络、训练和测试的部分，网上能找到很多有关资料，学起来很快，但我觉得最有意思的是我自己 实现了数据的加载，实现了自己定义的数据集。这部分网上能查到的资料比较少，大部分都是直接用给 定的接口做了。但比起直接用Pytorch已经弄好的数据接口读入，自己尝试着去读更有意思，也加深了 我对它的理解。特别是，一开始我以为CIFAR是需要转成灰度图的，查了很多资料应该怎么读入数据转 成灰度，做到后面发现CIFAR就应该用彩色图做，当时顿时觉得有点无奈，但回过头来看，我在查怎么转成灰度的时候，也加深了我对这个数据结构的了解。

另外有一点很有意思的是，CIFAR我最后是跑了50个epoch达到94%的准确率，其中汽车的识别率最 低，猫的识别率最高。但是一开始，我只跑了2个epoch试试看的时候，它的结果其实是这样的:



这个时候猫的正确率其实是很低的，而汽车却有比较好的识别效果。观察了一部分测试数据和结果之 后，我认为，除了训练随机性的差别以外，汽车的整体特征其实只需要比较少的训练次数就能大概得到，所以在epoch=2的时候就能有64%；而猫的图片很多都是一个模糊的比较小的形状，刚开始训练的时候是很难提取到特征的。但在epoch足够大的时候，比如我们最后用epoch=50，猫的特征能够捕捉 完全，正确率有很大提高，最后达到92%；汽车的一部分图片却很容易和卡车混起来，它们的某些特征是相似的，具体到一些特殊的图片更是如此.