LeNet 小作业

211870287 丁旭

2023 年 11 月 12 日

题目 1. 基于 Pytorch 实现 LeNet-5 ，并完成 CIFAR10 识别。可以尝试使用一些图像预处理技术（去噪，归一化，分割等），再使用神经网络进行特征提取。同时可以对训练过程进行可视化处理，分析训练趋势。需要提交的内容包括但不限于:

1. 可运行代码（LeNet.py），关键部分代码需要注释；
2. PDF 报告，报告中要有明确的实验过程说明、精度截图以及实验数据分析等。

搭建网络结构. LeNet-5 是一个经典的深度卷积神经网络，旨在解决手写数字识别问题。其输入为 32X32 的图像，构成主要为两个卷积层，两个池化层，以及三个全连接层组成。我以 LeNet 原始网络为骨架，使用 CIFAR-10 数据集，在 LeNet 原始骨架的基础上，并加以些许改进，最终在测试集达到了约 67% 的正确率。

实验步骤：

首先定义 LeNet 的网络结构

LeNet模型

输入：灰度图像，通道为1，尺寸为1×32×32

第一层：卷积层

LeNet-5模型接受的输入层大小是1×32x32。卷积层的过滤器的尺寸是5x5，深度（卷积核个数）为6，不使用全0填充，步长为1。则这一层的输出的尺寸为32-5+1=28，深度为6。本层的输出矩阵大小为6×28×28。

第二层：池化层

这一层的输入是第一层的输出，是一个6×28x28=4704的节点矩阵。本层采用的过滤器为2x2的大小，长和宽的步长均为2，所以本层的输出矩阵大小为6×14x14。

第三层：卷积层

本层的输入矩阵大小为6×14x14，使用的过滤器大小为5x5，深度为16。本层不使用全0填充，步长为1。本层的输出矩阵大小为16×10x10。

第四层：池化层

本层的输入矩阵大小是16×10x10，采用的过滤器大小是2x2，步长为2，本层的输出矩阵大小为16×5x5。

第五层：全连接层（LeNet5有3个全连接层，输出维度分别是120，84，10）

本层的输入矩阵大小为16×5x5。将此矩阵中的节点拉成一个向量，那么这就和全连接层的输入一样了，本层的输出节点个数为120。

第六层：全连接层

本层的输入节点个数为120个，输出节点个数为84个。

第七层：全连接层

本层的输入节点为84个，输出节点个数为10个。

```python
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        # 卷积层1，输入图像深度=3，输出图像深度=16，卷积核大小=5*5，卷积步长=1;16表示输出维度，也表示卷积核个数
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=1)
        # 池化层1，采用最大池化，区域集大小=2*2.池化步长=2
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # 卷积层2
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1)
        # 池化层2
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # 全连接层1：输入大小=32*5*5，输出大小=120
        self.fc1 = nn.Linear(32*5*5, 120)
        # 全连接层2
        self.fc2 = nn.Linear(120, 84)
        # 全连接层3
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))  # input(3, 32, 32) output(16, 28, 28)
        x = self.pool1(x)  # output(16, 14, 14)
        x = F.relu(self.conv2(x))  # output(32, 10, 10)
        x = self.pool2(x)  # output(32, 5, 5)
        x = x.view(-1, 32 * 5 * 5)  # output(32*5*5)
        x = F.relu(self.fc1(x))  # output(120)
        x = F.relu(self.fc2(x))  # output(84)
        x = self.fc3(x)  # output(10)
        return x
```

图 1: 网络结构

数据集下载

然后获取 CIFAR-10 数据集以及测试集，对图像做了归一化处理：

```python
'''2.下载数据集并查看部分数据'''
# transforms.Compose()函数将两个函数拼接起来。
# （ToTensor()：把一个PIL.Image转换成Tensor，Normalize()：标准化，即减均值，除以标准差）
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
# 训练集：下载CIFAR10数据集，如果没有事先下载该数据集，则将download参数改为True
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=False, transform=transform)
# 用DataLoader得到生成器，其中shuffle：是否将数据打乱；
# num_workers表示使用多进程加载的进程数，0代表不使用多进程
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True, num_workers=0)

# 测试集数据下载
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=False, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=False, num_workers=0)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

图 2: 获取并预处理数据集

进行模型训练

对于多分类问题，应该使用Softmax函数，这里CIFAR10数据集属于多分类，却使用了交叉熵损失函数，是因为进入CrossEntropyLoss()函数内部就会发现其中包含了Softmax函数

```python
# 对于多分类问题，应该使用Softmax函数，这里CIFAR10数据集属于多分类，却使用了交叉熵损失函数，
# 是因为进入CrossEntropyLoss()函数内部就会发现其中包含了Softmax函数
loss_function = nn.CrossEntropyLoss()  # 使用交叉熵损失函数
# 优化器选择Adam，学习率设为0.001
optimizer = optim.Adam(net.parameters(), lr=0.001)
```

```python
# 初始化损失值列表
loss_list = []
acc_list = []
for epoch in range(10):  # 整个迭代10轮
    running_loss = 0.0  # 初始化损失函数值loss=0
    correct = 0
    total = 0
    for i, data in enumerate(trainloader, start=0):
        # 获取训练数据
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)  # 将数据及标签传入GPU/CPU

        # 权重参数梯度清零
        optimizer.zero_grad()

        # 正向及反向传播
        outputs = net(inputs)
        loss = loss_function(outputs, labels)
        loss.backward()
        optimizer.step()

        # 显示损失值
        running_loss += loss.item()
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %(epoch + 1, i + 1, running_loss / 2000))

            # 记录损失值
            loss_list.append(running_loss / 2000)
            acc_list.append(100 * correct / total)

            running_loss = 0.0
            correct = 0
            total = 0

print('Finished Training')
```

图 3: 损失函数与优化器

采用训练轮数 epoch 为 10，绘制正确率和损失率图

```
# 绘制损失函数随时间的变化曲线
plt.subplot(2, 1, 1)
plt.plot(loss_list)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training Loss')

# 绘制准确率随时间的变化曲线
plt.subplot(2, 1, 2)
plt.plot(acc_list)
plt.xlabel('Iteration')
plt.ylabel('Accuracy')
plt.title('Training Accuracy')
plt.show()
```

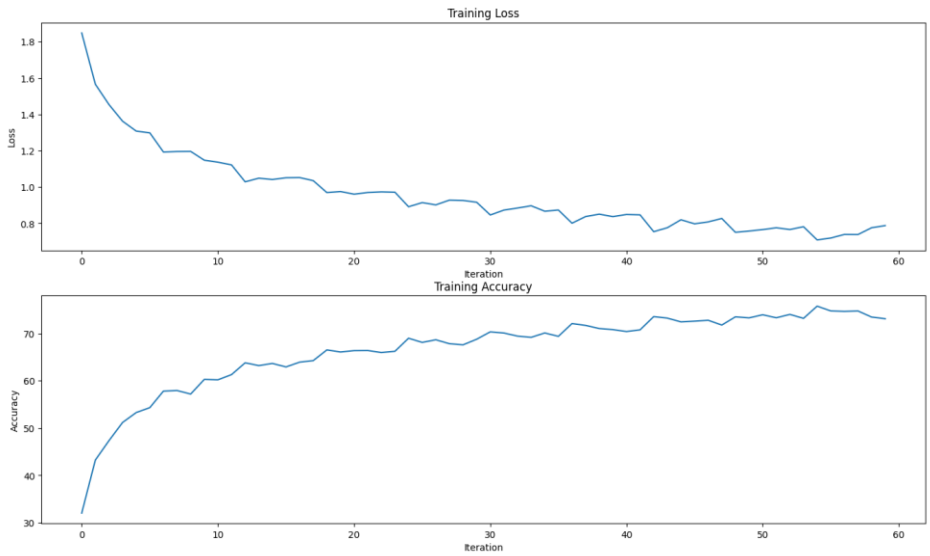

图 4: 测试集上的准确率和loss

```
Accuracy of the network on the 10000 test images: 67 %
Accuracy of plane : 76 %
Accuracy of   car : 76 %
Accuracy of  bird : 46 %
Accuracy of   cat : 37 %
Accuracy of  deer : 62 %
Accuracy of   dog : 58 %
Accuracy of  frog : 86 %
Accuracy of horse : 70 %
Accuracy of  ship : 79 %
Accuracy of truck : 77 %
```

图 5: 每一类的正确率

实验数据分析：

通过图像可以看到，训练过程中，从输出结果可以看出，随着训练的进行，损失值在逐渐下降，而准确率在逐渐上升。我的模型在训练过程中逐渐收敛并取得了一定的效果。

测试准确率：在测试集上，我的模型的准确率为大约70%左右。在调整学习率，调整训练轮数 (epoch) 等其他参数时未有明显的精度提升，反而还可能导致精度下降或不收敛的情况。后通过网上查找相关精度提升策略，在网络结构中加入 Relu 激活函数以及在 SGD 优化器中加入动量为 0.9 的设置，使训练精度在原始的 LeNet 模型精度下提升了 10% 左右。

类别准确率：对于每个类别的图像，可以查看模型在该类别上的分类准确率（图五）。Cat的分类效果比较差，有可能需要针对性地调整模型或增加类别样本数量。Frog的分类效果最好。