实验三

CIFAR10图像分类

2022年10月2日

1. 读取数据文件

使用torchvision.datasets的CIFAR10数据集，训练集进行旋转和裁剪操作。用DataLoader的方式分批读取

读取数据文件

In [2]:
```
train_trans = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),    # 随机旋转
    transforms.RandomCrop(32, padding=4),      # 放大后随机裁剪到32*32
    transforms.ToTensor(),                     # 转为tensor
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))    # 归一化
])

test_trans = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])
trainset = torchvision.datasets.CIFAR10(root="./data",train=True,download=True,transform=train_trans)
testset = torchvision.datasets.CIFAR10(root="./data",train=False,download=True,transform=test_trans)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

In [3]:
```
# 用dataloader的方式分批读取
train_loader = DataLoader(dataset=trainset, batch_size=20, shuffle=True, num_workers=0)
test_loader = DataLoader(dataset=testset, batch_size=20, shuffle=False, num_workers=0)

# gpu 加速
device = torch.device('cpu')
```

实现训练和测试函数

In [4]:
```
# 每个epoch的训练结果
def train_epoch(model,dataloader):
    model.train()
    running_loss = 0
    for i,(img,label) in enumerate(tqdm(dataloader)):
        img,label = img.to(device),label.to(device)    # 将数据读入gpu
        out = model(img)                               # 模型预测结果
        optimizer.zero_grad()
        loss = criterion(out,label)                    # 计算损失
        loss.backward()                                # 反向传播更新梯度
        optimizer.step()
        running_loss += loss.item() * img.size(0)
    return running_loss / len(dataloader.dataset)
```

In [5]:
```
def test(model,dataloader):
    model.eval()
    total = 0
    correct = 0
    for i,(img,label) in enumerate(tqdm(dataloader)):
        img,label = img.to(device),label.to(device)
        out = model(img)
        _, pred = torch.max(out,1)                     # 计算正确率
        correct += (pred == label).sum().item()
        total += label.size(0)
    acc = 100.0 * correct / total
    return acc
```

In [6]:
```
def train(model, n_epochs):
    loss_arr = []
    acc_arr = []
    for epoch in range(n_epochs):
        loss = train_epoch(model, train_loader)
        acc = test(model, train_loader)
        print('Epoch {}, Train Loss {}, Train Acc {}'.format(epoch + 1, loss, acc))
        loss_arr.append(loss)
        acc_arr.append(acc)
    return loss_arr, acc_arr
```

2. MLP模型搭建

搭建了具有3个全连接层的MLP，隐含层结点分别为1024， 256，用ReLU作为隐含层的激活函数。

MLP模型搭建

In [7]:
```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = torch.nn.Linear(3 * 32 * 32, 1024)
        self.fc2 = torch.nn.Linear(1024, 256)
        self.fc3 = torch.nn.Linear(256, 10)

    def forward(self, x):
        x = x.view(-1, 3 * 32 * 32)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

使用CrossEntropyLoss作为损失函数，使用学习率为0.01，衰退权重为0.0001的SGD优化器进行训练。50次迭代后的预测正确率为54.9%。

In [8]:
```
n_epochs = 50
mlp = MLP().to(device)
criterion = nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.SGD(params=mlp.parameters(), lr=0.01, weight_decay=0.0001)
mlp_loss, mlp_acc = train(mlp, n_epochs)
print('Test Acc: {}'.format(test(mlp, test_loader)))
```

```
100%|██████████| 2500/2500 [00:13<00
:00, 189.28it/s]
100%|██████████| 2500/2500 [00:06<00
:00, 389.15it/s]
Epoch 49, Train Loss 1.073707194006443, Train Acc 62.678
100%|██████████| 2500/2500 [00:13<00
:00, 190.43it/s]
100%|██████████| 2500/2500 [00:06<00
:00, 388.42it/s]
Epoch 50, Train Loss 1.072228419649601, Train Acc 62.744
100%|██████████| 500/500 [00:00<00
:00, 505.34it/s]
Test Acc: 54.9
```

3. CNN模型搭建

搭建了具有3个卷积层和池化层，以及3个全连接层的卷积神经网络

输入的是3 * 32 * 32的数据

第一个卷积层使用6 * 5 * 5的卷积核，输出6 * 28 * 28

第一个池化层进行2 * 2的最大池化，输出6 * 14 * 14

第二个卷积层使用16 * 3 * 3的卷积核，输出16 * 12 * 12

第二个池化层进行2*2的最大池化，输出16 * 6 * 6

第三个卷积层使用64 * 2 * 2的卷积核，输出64 * 4 * 4

第三个池化层进行2 * 2的最大池化，输出64 * 2 * 2

后面接入三个全连接层，隐含层使用ReLU作为激活函数。

CNN模型搭建

In [9]:
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)      # 6 * 28 * 28
        self.pool1 = nn.MaxPool2d(2)         # 6 * 14 * 14
        self.conv2 = nn.Conv2d(6, 16, 3)     # 16 * 12 * 12
        self.pool2 = nn.MaxPool2d(2)         # 16 * 6 * 6
        self.conv3 = nn.Conv2d(16, 64, 3)    # 64 * 4 * 4
        self.pool3 = nn.MaxPool2d(2)         # 64 * 2 * 2
        self.fc1 = nn.Linear(64 * 2 * 2 , 1024)
        self.fc2 = nn.Linear(1024, 256)
        self.fc3 = nn.Linear(256, 10)


    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2((self.conv2(x)))
        x = self.pool3(F.relu(self.conv3(x)))
        x = x.view(-1, 64 * 2 * 2)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

使用CrossEntropyLoss作为损失函数，使用学习率为0.01，衰退权重为0.0001的SGD优化器进行训练。50次迭代后的预测正确率为74.03%。

In [10]:
```
cnn = CNN().to(device)
criterion = nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.SGD(params=cnn.parameters(), lr=0.01, weight_decay=0.0001)
cnn_loss, cnn_acc = train(cnn, n_epochs)
print('Test Acc: {}'.format(test(cnn, test_loader)))
```

```
100%|| 2500/2500 [00:18<00
:00, 137.19it/s]
100%|| 2500/2500 [00:08<00
:00, 306.63it/s]
Epoch 49, Train Loss 0.6774460505485534, Train Acc 76.284
100%|| 2500/2500 [00:18<00
:00, 136.23it/s]
100%|| 2500/2500 [00:08<00
:00, 307.28it/s]
Epoch 50, Train Loss 0.6712483007788658, Train Acc 77.026
100%|| 500/500 [00:01<00
:00, 374.38it/s]
Test Acc: 74.03
```

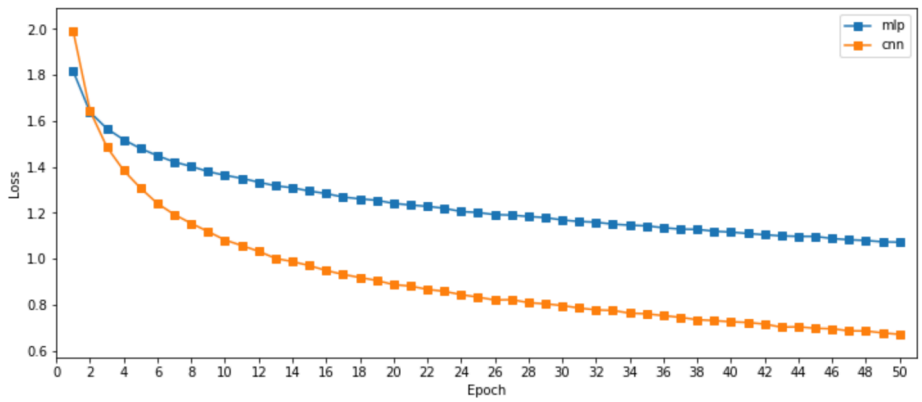
3. 绘制变化曲线

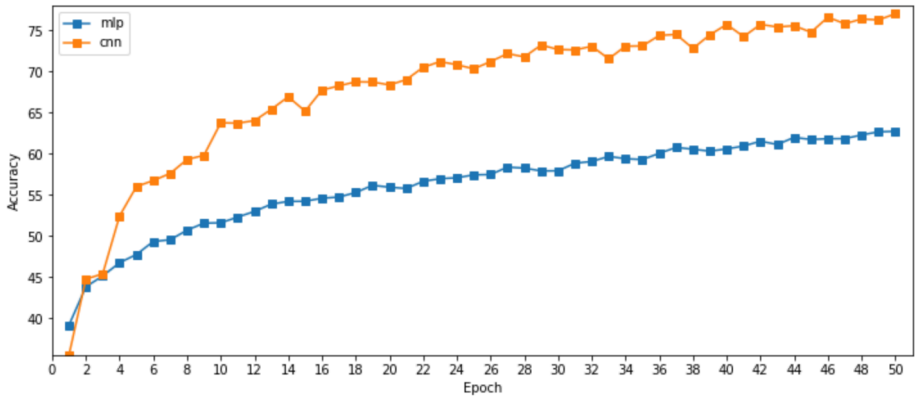
使用前两个实验的代码

```python
import matplotlib.pyplot as plt
def plot_loss_and_acc(loss_and_acc_dict):
    # visualize loss curve
    plt.figure(figsize=(12, 5))
    min_loss, max_loss = 100.0, 0.0
    for key, (loss_list, acc_list) in loss_and_acc_dict.items():
        min_loss = min(loss_list) if min(loss_list) < min_loss else min_loss
        max_loss = max(loss_list) if max(loss_list) > max_loss else max_loss
        num_epoch = len(loss_list)
        plt.plot(range(1, 1 + num_epoch), loss_list, '-s', label=key)
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.xticks(range(0, num_epoch + 1, 2))
    plt.axis([0, num_epoch + 1, min_loss - 0.1, max_loss + 0.1])
    plt.show()
    # visualize acc curve
    plt.figure(figsize=(12, 5))
    min_acc, max_acc = 100.0, 0.0
    for key, (loss_list, acc_list) in loss_and_acc_dict.items():
        min_acc = min(acc_list) if min(acc_list) < min_acc else min_acc
        max_acc = max(acc_list) if max(acc_list) > max_acc else max_acc
        num_epoch = len(acc_list)
        plt.plot(range(1, 1 + num_epoch), acc_list, '-s', label=key)
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.xticks(range(0, num_epoch + 1, 2))
    plt.axis([0, num_epoch + 1, min_acc, max_acc + 1])
    plt.show()
```

从结果中可以看出，CNN的loss更小，而且正确率也比MLP的要高20%

```python
plot_loss_and_acc({
    "mlp": [mlp_loss, mlp_acc],
    "cnn": [cnn_loss, cnn_acc]
})
```





4. 调整优化器

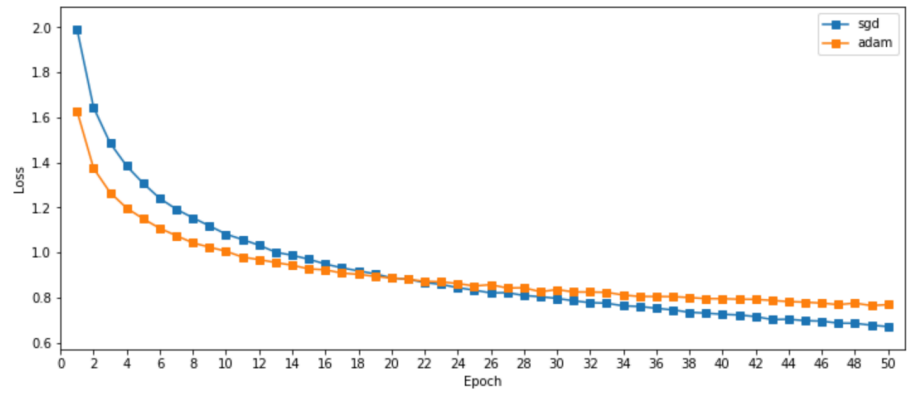
使用学习率为0.001的Adam优化器训练

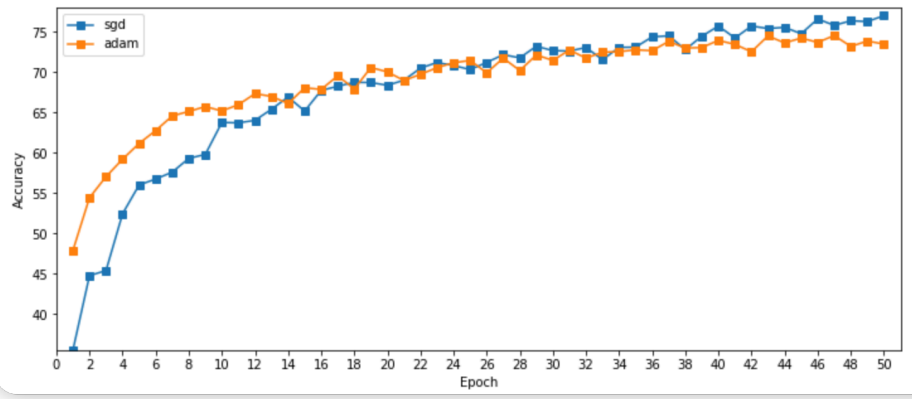
```
cnn = CNN().to(device)
optimizer = torch.optim.Adam(params=cnn.parameters(), lr=0.001)
cnn_adam_loss, cnn_adam_acc = train(cnn, n_epochs)
```

```
:00, 110.89it/s]
100%|██████████| 2500/2500 [00:08<00
:00, 298.77it/s]

Epoch 48, Train Loss 0.7755873042285443, Train Acc 73.206

100%|██████████| 2500/2500 [00:22<00
:00, 110.44it/s]
100%|██████████| 2500/2500 [00:08<00
:00, 298.59it/s]

Epoch 49, Train Loss 0.7645300450325012, Train Acc 73.874

100%|██████████| 2500/2500 [00:22<00
:00, 111.77it/s]
100%|██████████| 2500/2500 [00:08<00
:00, 299.41it/s]

Epoch 50, Train Loss 0.7688073013603687, Train Acc 73.496
```

与使用SGD优化器对比为：

```
plot_loss_and_acc({
    "sgd": [cnn_loss, cnn_acc],
    "adam": [cnn_adam_loss, cnn_adam_acc]
})
```





可以看出一开始Adam优化器的损失下降的较快，但是随着epoch的增加，adam的损失则变化得较小，而SGD的正确率也逐渐超过Adam。

5. 调整网络层数

增加一个全连接层，并在全连接层之间添加了一个dropout层。

```
class CNNNew(nn.Module):
    def __init__(self):
        super(CNNNew, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)       # 6 * 28 * 28
        self.pool1 = nn.MaxPool2d(2)          # 6 * 14 * 14
        self.conv2 = nn.Conv2d(6, 16, 3)      # 16 * 12 * 12
        self.pool2 = nn.MaxPool2d(2)          # 16 * 6 * 6
        self.conv3 = nn.Conv2d(16, 64, 3)     # 64 * 4 * 4
        self.pool3 = nn.MaxPool2d(2)          # 64 * 2 * 2
        self.fc1 = nn.Linear(64 * 2 * 2 , 2048)
        self.fc2 = nn.Linear(2048, 512)
        self.fc3 = nn.Linear(512, 256)
        self.fc4 = nn.Linear(256, 10)


    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2((self.conv2(x)))
        x = self.pool3(F.relu(self.conv3(x)))
        x = x.view(-1, 64 * 2 * 2)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.dropout(x, 0.3)                 # 增加dropout层
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return x
```

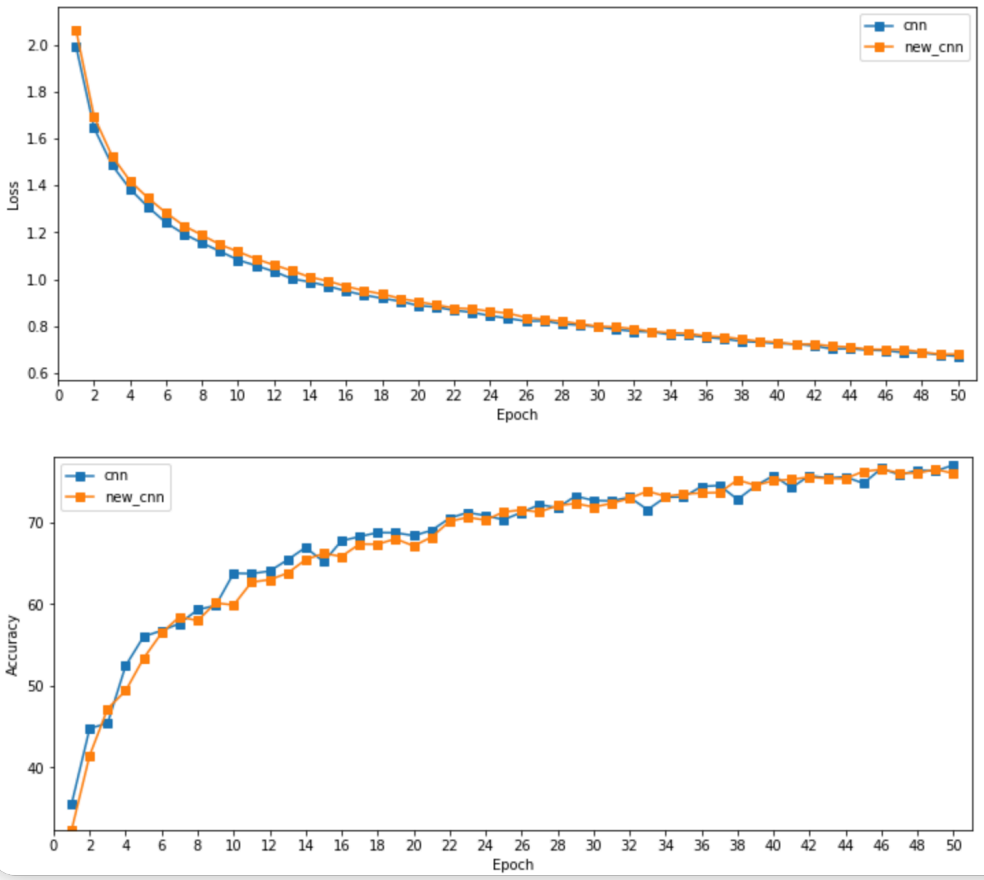
使用CrossEntropyLoss作为损失函数，使用学习率为0.01，衰退权重为0.0001的SGD优化器进行训练。50次迭代后的预测正确率为73.13%。

```
cnn = CNNNew().to(device)
criterion = nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.SGD(params=cnn.parameters(), lr=0.01, weight_decay=0.0001)
cnn_new_loss, cnn_new_acc = train(cnn, n_epochs)
print('Test Acc: {}'.format(test(cnn, test_loader)))
```

```
100%|██████████| 2500/2500 [00:27<0
0:00, 91.29it/s]
100%|██████████| 2500/2500 [00:10<00
:00, 244.78it/s]

Epoch 49, Train Loss 0.6798847847402095, Train Acc 76.496

100%|██████████| 2500/2500 [00:27<0
0:00, 91.15it/s]
100%|██████████| 2500/2500 [00:10<00
:00, 243.20it/s]

Epoch 50, Train Loss 0.6801970268666744, Train Acc 75.998

100%|██████████| 500/500 [00:01<00
:00, 283.65it/s]

Test Acc: 73.13
```

与原来CNN的对比中，可以看到两者曲线基本一致，并没有看出增加层数带来的效果。

```
plot_loss_and_acc({
    "cnn": [cnn_loss, cnn_acc],
    "new_cnn": [cnn_new_loss, cnn_new_acc]
})
```



6. 修改隐含层激活函数

将隐含层激活函数替换为Tanh

```
class CNNTanh(nn.Module):
    def __init__(self):
        super(CNNTanh, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)       # 6 * 28 * 28
        self.pool1 = nn.MaxPool2d(2)          # 6 * 14 * 14
        self.conv2 = nn.Conv2d(6, 16, 3)      # 16 * 12 * 12
        self.pool2 = nn.MaxPool2d(2)          # 16 * 6 * 6
        self.conv3 = nn.Conv2d(16, 64, 3)     # 64 * 4 * 4
        self.pool3 = nn.MaxPool2d(2)          # 64 * 2 * 2
        self.fc1 = nn.Linear(64 * 2 * 2 , 1024)
        self.fc2 = nn.Linear(1024, 256)
        self.fc3 = nn.Linear(256, 10)


    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2((self.conv2(x)))
        x = self.pool3(F.relu(self.conv3(x)))
        x = x.view(-1, 64 * 2 * 2)
        x = F.tanh(self.fc1(x))
        x = F.tanh(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
cnn = CNNTanh().to(device)
criterion = nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.SGD(params=cnn.parameters(), lr=0.01, weight_decay=0.0001)
cnn_tanh_loss, cnn_tanh_acc = train(cnn, n_epochs)
print('Test Acc: {}'.format(test(cnn, test_loader)))
```

```
100%|██████████| 2500/2500 [00:21<00
:00, 118.18it/s]
100%|██████████| 2500/2500 [00:08<00
:00, 280.71it/s]

Epoch 49, Train Loss 0.7560197077989578, Train Acc 74.762

100%|██████████| 2500/2500 [00:21<00
:00, 117.66it/s]
100%|██████████| 2500/2500 [00:08<00
:00, 280.30it/s]

Epoch 50, Train Loss 0.7479605409920216, Train Acc 74.364

100%|██████████| 500/500 [00:01<00
:00, 337.27it/s]

Test Acc: 73.51
```

对比中，可以看出，ReLU作为激活函数的效果要更好一点。

```
plot_loss_and_acc({
    "relu": [cnn_loss, cnn_acc],
    "tanh": [cnn_tanh_loss, cnn_tanh_acc]
})
```