人工智能实验 CNN+cifar10报告

郭凯 20354034
王浩祯 20354248
曾陆豪 20354192
陈清桦 20354014
陈柔柔 20354015
陈光燕 20354007

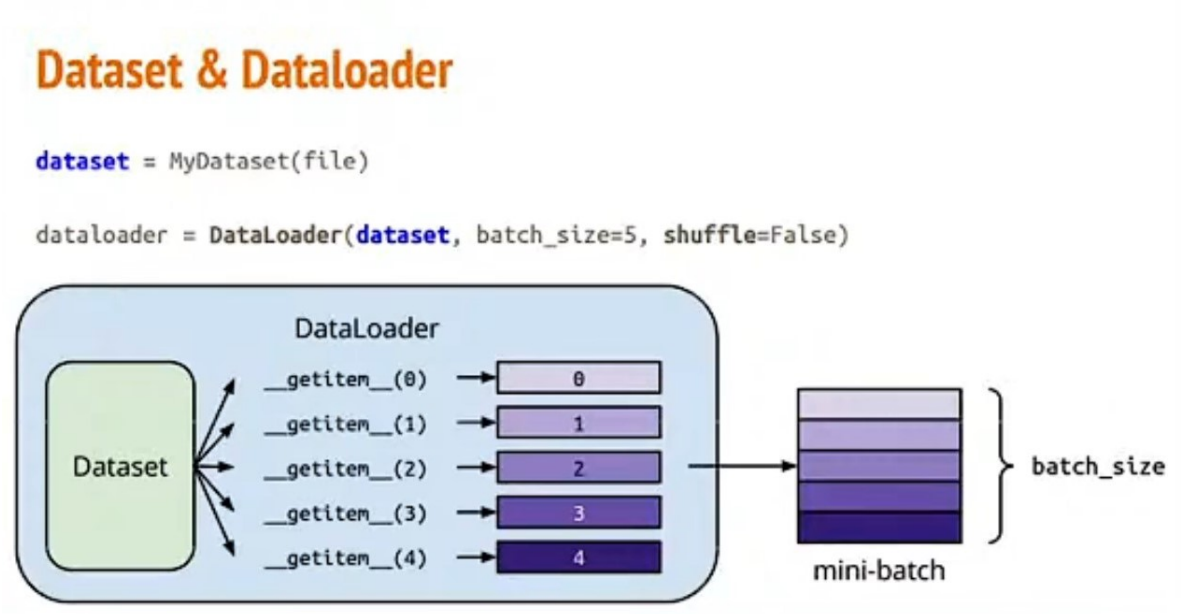
使用普通CNN进行cifar10图像分类

一、导入数据与数据预处理

Dataloader导入数据集

在Pytorch中，提供了Dataset和Dataloader两个工具包来构造数据加载器，用于数据集的加载。Dataset用于定义数据读取和储存的方式，而Dataloader结合了数据集和取样器，用来把训练数据分成多个小组，此函数每次抛出一组数据。直至把所有的数据都抛出。就是做一个数据的初始化。

> 需要注意的是，Dataset类只相当于一个打包工具，包含了数据的地址。真正把数据读入内存的过程是由Dataloader进行批迭代输入的时候进行的。



Dataset将数据打包，Dataloader则是调用getitem函数根据列表中每张图片的地址读取图片。

在本次实验中，我们通过继承和重写Dataset和DataLoader两个类来构造我们自己的数据加载器。dataloader.py (见附件)是我们构造的数据加载器，在训练网络时导入该模块即可。

```
class my_dataset(Dataset):
    def __init__(self, store_path, split, name,
data_transform=None):
        self.store_path = store_path
        self.split = split
        self.name = name
        self.transforms = data_transform
        self.img_list = []  # 储存每张图片的路径
        self.label_list = []  # 储存每张图片的类别
        for file in glob.glob(self.store_path + '/' + split +
'/*png'):
            cur_path = file.replace('\\', '/')  # 每张图片的路径,用/
替代路径中的\\
            cur_label = cur_path.split('_')[-1].split('.png')[0]
 # 获取每张图片的类别名
            self.img_list.append(cur_path)
            self.label_list.append(self.name[cur_label])  # name
是一个字典，将每个类别用一个数字代替

    def __getitem__(self, item):
        img = Image.open(self.img_list[item]).convert('RGB')
        if self.transforms is not None:
            img = self.transforms(img)  # 转换
        label = self.label_list[item]
```

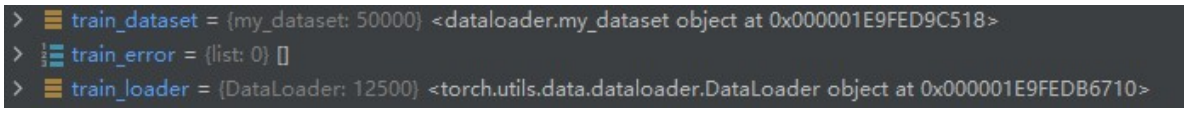
```
        return img, label

    def __len__(self):
        return len(self.img_list)

# 训练集
split = 'train'
train_dataset = my_dataset(store_path, split, label, transform)
train_loader = DataLoader(train_dataset, batch_size=4,
shuffle=False)
```

我们通过重写和继承Dataset类来定义自己的数据加载器my_dataset。my_dataset将数据打包，将每张图片的地址放入列表中。Dataloader则是调用getitem函数根据列表中每张图片的地址读取图片。其中Dataloader中batch_size=4表示每次输入网络有4张图片。

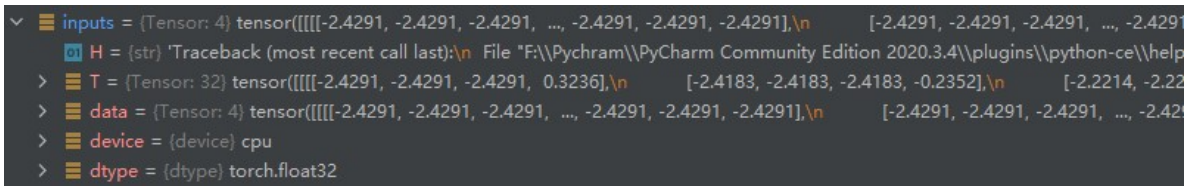
store_path是数据集储存路径，split表面是训练集还是测试集，label是储存分类种类的一个字典，transform是数据预处理方式(下文讲)。

导入数据的返回形式

我们使用train_dataset来表示读取的数据集，而train_loader表示数据加载器。在debug中我们可以看出，train_dataset读取了50000张图片，而由于加载器中我们定义batch_size=4，即每次输入网络的图片有4张，因此在train_loader中有12500组，每组有4张图片。



接着，我们遍历train_loader，将其中的每一组作为网络的输入



可以看到，这里inputs是一个长度为4的Tensor，表示有4张图片。

数据预处理

我们采用torchvision中的transforms机制来对数据进行预处理。transforms.Compose()，是将一系列的transforms步骤有序组合，实现时按照这些方法依次对图像操作。简单来说就是将所有预处理的步骤进行一个打包，在上面dataset中导入transform即可对读取的数据集进行一个统一的预处理。

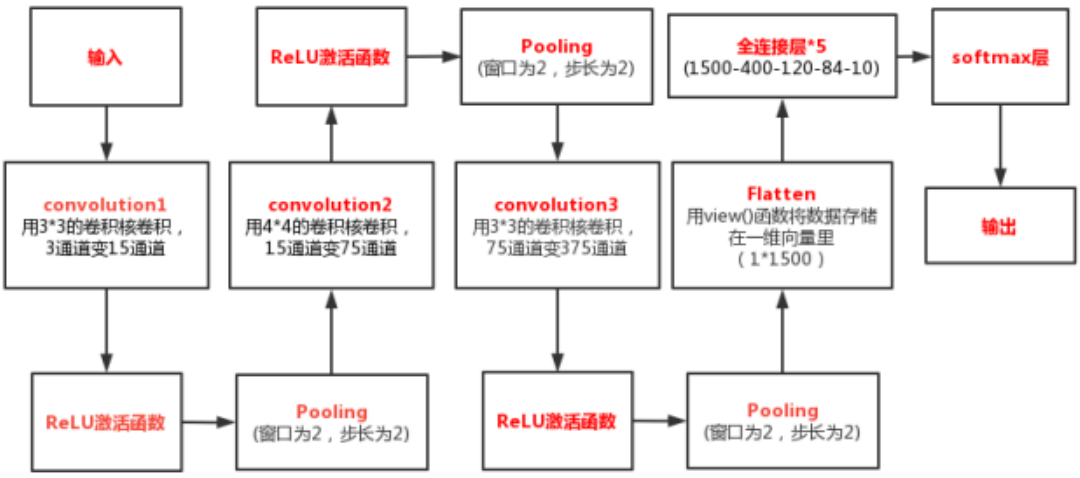
```
transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),  # 先四周填充0，在吧图像随机裁剪成32*32
    transforms.RandomHorizontalFlip(),  # 图像一半的概率翻转，一半的概率不翻转
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),  # R,G,B每层的归一化用到的均值和方差
])
```

这里我们将读取的图片进行数据增广，先进行一个填充和裁剪的步骤，然后概率翻转来模拟环境，用于提高学习能力，然后讲图片转化为Tensor形式，最后数据归一化。

ToTensor()能够把灰度范围从0-255变换到0-1之间，而后面的transform.Normalize()用均值和标准差对张量图像进行归一化,把0-1变换到(-1,1)。RandomCrop可以减弱背景(或噪音)因子的权重，且使模型面对缺失值不敏感，也就可以产生更好的学习效果，增加模型稳定性。

二、CNN网络结构

在本次实验中，我们采用3个卷积层，3个池化层，以及5个全连接层和1个softmax层。网络结构如下：

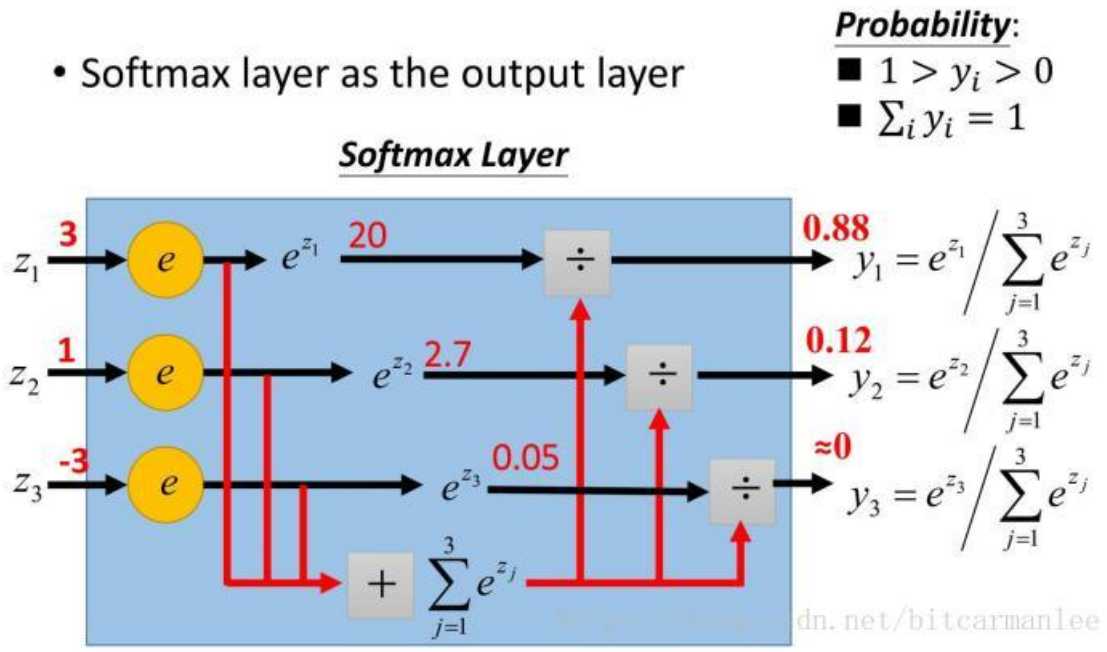


网络中，我们采用的是ReLu激活函数，第1个卷积层有15个卷积核，第2个卷积层有75个卷积核，第3个卷积层有375个卷积核。池化层采样大小为2*2。全连接层神经元个数为1500-400-120-84-10。

输出结果

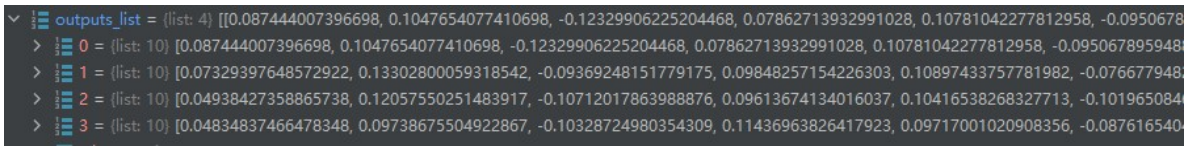
最终的结果是经过Softmax层的输出结果。由于我们最后是要分10类，因此全连接层的最后一层也要是10个神经元，最终的输出结果是一个1*10的向量。而Softmax层就是将该向量中的10个数值间隔放大，将这10个数转化为一个概率，表示分到该类的概率是多少。

例如：假设最终的输出结果为：[0.12, 0.64, 0.24]，那么就表示这张图片分到第0，1，2类的概率分别为12%、64%、24%。结果Softmax层后的向量，其中所有元素相加的和始终为1。



但是，在我们的代码中，实际上的网络结构并没有经过Softmax层，而是把Softmax层并入损失函数内。这样做的好处是，减小计算难度，加快计算速度。在计算误差时我们经过了Softmax层，但网络的输出值并不是Softmax层的结果，而是经过全连接层后直接输出了。由于在损失函数中我们加入了Softmax层，因此最终结果对分类效果并无影响。

我们来看看图片结果CNN网络后的输出结果：



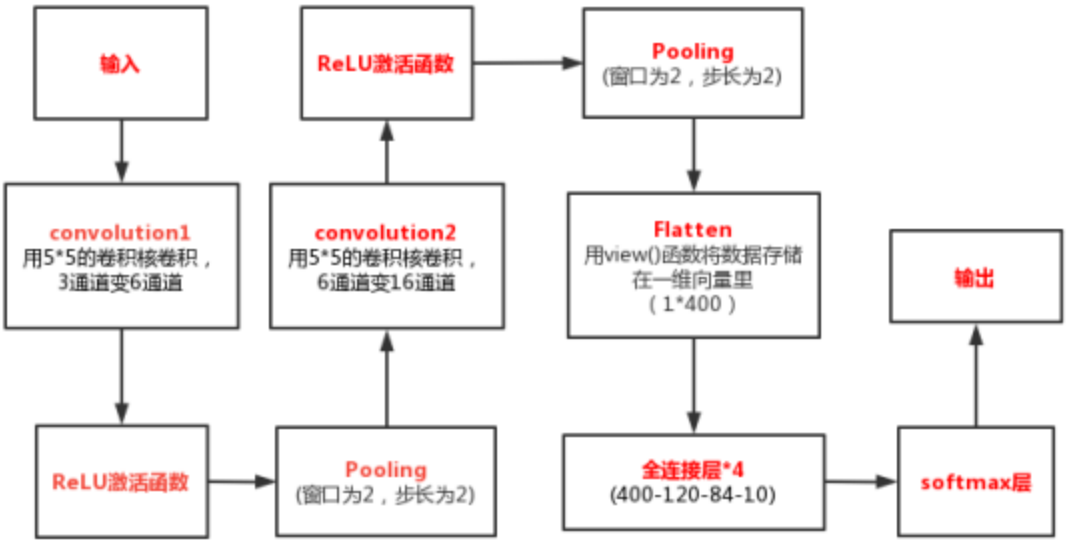
我们将输出的outputs转化为一个列表list，可以看出这是一个4*10的列表。由于我们上面batch_size=4，每一组数据有4张图片，因此输出的结果也是4张图片的结果。而列表每一行有10个数据，分别表示10个类。我们取每一行的最大值所在的位置，就代表该图片所分的类。例如，第一行中最大的数是第5个，表示这张图片分到第5类，也就是 'deer' 类。

```
label = {'airplane': 0, 'automobile': 1, 'bird': 2, 'cat': 3,
'deer': 4, 'dog': 5,'frog': 6, 'horse': 7, 'ship': 8,'truck': 9}
 # 类别标签
```

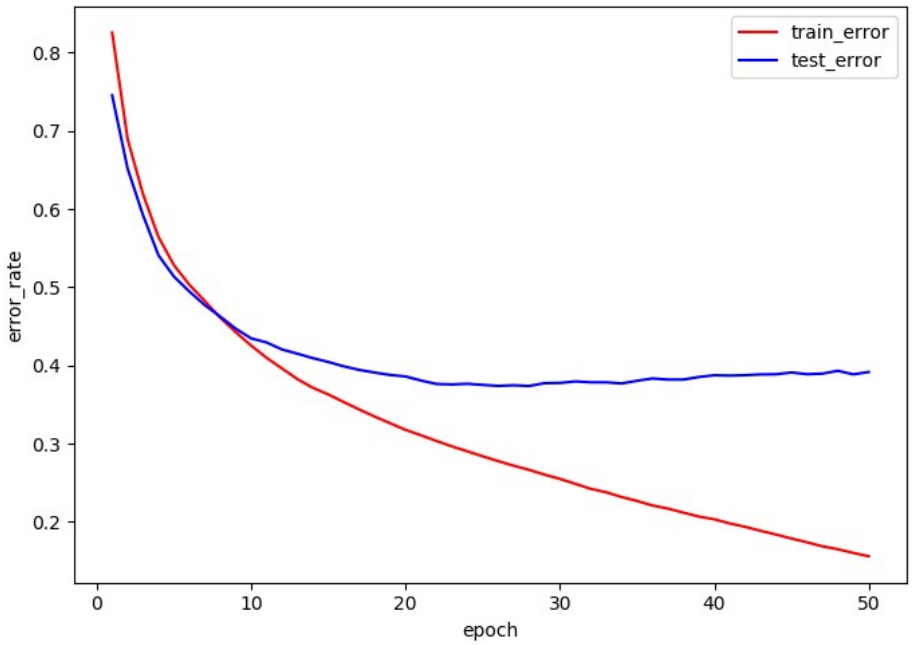
三、结果分析和改进方法

结果分析

我们最初采用的CNN网络结构如下图所示：



一共有2个卷积层、2个池化层、4个全连接层。第一个卷积层有6个5*5大小的卷积核，第二个卷积层有16个5*5大小的卷积核。全连接层有4层，分别有400-120-84-10个神经元。我们设定epoch=50，以错误率为模型评定的指标，错误率=分类错误个数/图片总数。得到统计图如下：



图中，蓝色线为测试集的错误率，红色线为训练集的错误率。可以看出，在epoch=12左右，测试集的错误率已经接近饱和了，难以再下降。最终错误率维持在36%左右，也就是说准确率在64%左右。

改进方法

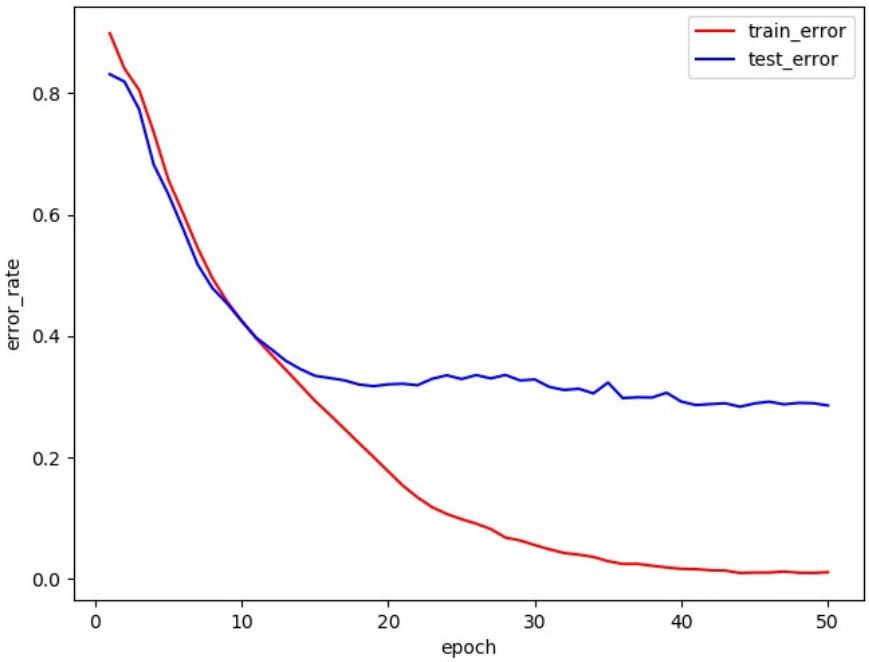
我们尝试从两个方向进行模型改进：

一、改变网络结构，在保证准确率的同时使其轻量化。

二、对数据预处理进行修改，采用数据增广的方法。

不同网络结构对结果的影响

我们尝试将网络的全连接层和卷积层扩大，由原来的两层卷积层和池化层增加到三层，原来的4层全连接层增加到5层。最终训练结果统计如下：



相比于一开始的网络结构，我们增加网络的层数使得错误率下降了5%左右，虽然有提升，但并不太明显。从这里我们可以看出，网络的复杂程度对最终的准确率提升带来的效果有限。所以，我们换一个思路，改变数据预处理的步骤，对图片进行数据增广。

不同数据预处理形式对结果的影响

在一开始，我们采用的transform仅仅只有归一化和转化为Tensor两个步骤。

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

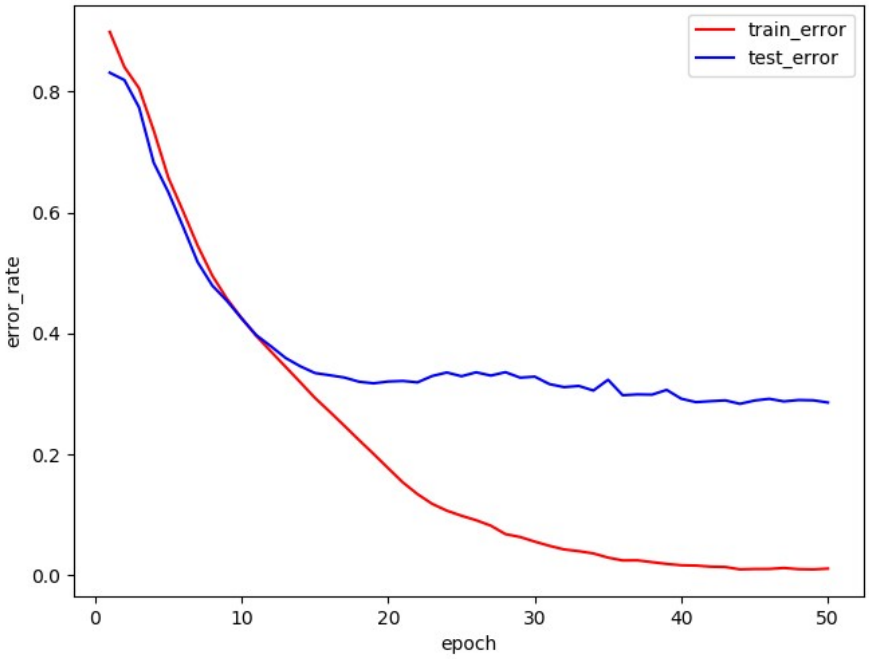
后来，我们仔细观察了cifar10的数据集，发现每张图片都是32*32大小，并且每张图片或多或少都存在些许其他元素的干扰。这时，我们想起了课上讲过的数据增广。数据增广，可以有效**避免复杂网络训练时陷入过拟合，训练得到的模型会更鲁棒，同时能显著提高数据质量。**

于是，我们改变了transform的步骤：

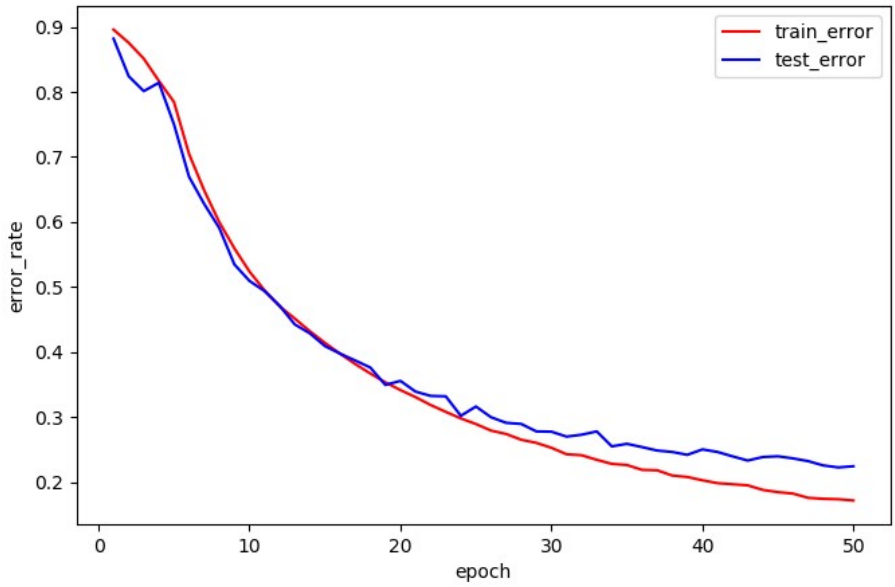
```
transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),  # 先四周填充0，在吧图像随机裁剪成32*32
    transforms.RandomHorizontalFlip(),  # 图像一半的概率翻转，一半的概率不翻转
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)), # R,G,B每层的归一化用到的均值和方差
])
```

先对每张图片周围填充0，然后随机裁剪成32*32的大小。同时对每张图片进行概率翻转，然后转化成Tensor再归一化。RandomCrop可以减弱背景(或噪音)因子的权重，且使模型面对缺失值不敏感，也就可以产生更好的学习效果，增加模型稳定性。

在进行数据增广之前，我们训练的网络误差为：

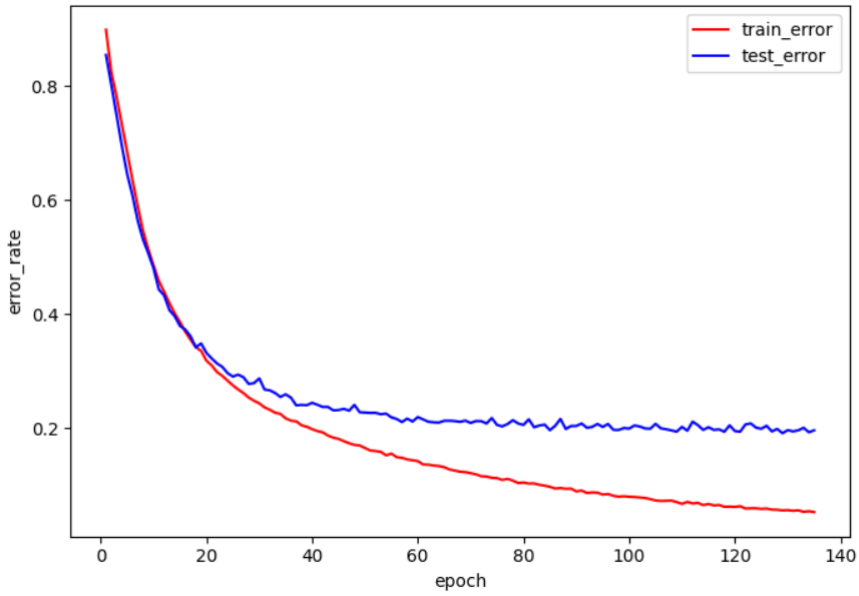


进行数据增广之后，训练的网络误差为：



对比两者，可以发现数据增广的步骤对我们最终的准确率有着非常大的提升，错误率从原来的32%左右降低到如今的22%左右。

然后，我们将epoch调大，最终训练得模型如下：



可以看见，模型训练误差在5%左右，测试误差在19%左右，准确率高达80%

我们将最后得到的模型放入测试接口中，使用测试集对模型进行测试，所得结果如下：

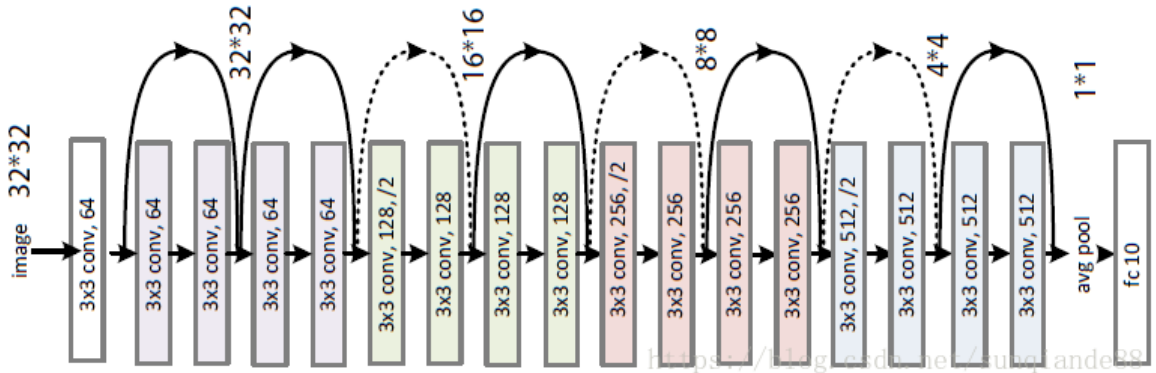
```
cuda
测试样本总数: 10000

CNN分类正确个数: 8016
CNN分类错误个数: 1984
CNN分类准确率: 0.801600
CNN分类错误率: 0.198400
损失函数值loss: 0.202561
```

我们最终的测试集准确率为80.16%

改进方法：采用ResNet18进行cifar10图像分类

网络结构

本次实验中，我们尝试使用ResNet来对cifar10进行分类。考虑到训练的困难程度(GPU受限)，以及我们想尽可能的使网络轻量化，因此我们决定采用ResNet18对cifar10进行分类。这里的18指17+1，表示有18个卷积层和1个全连接层。



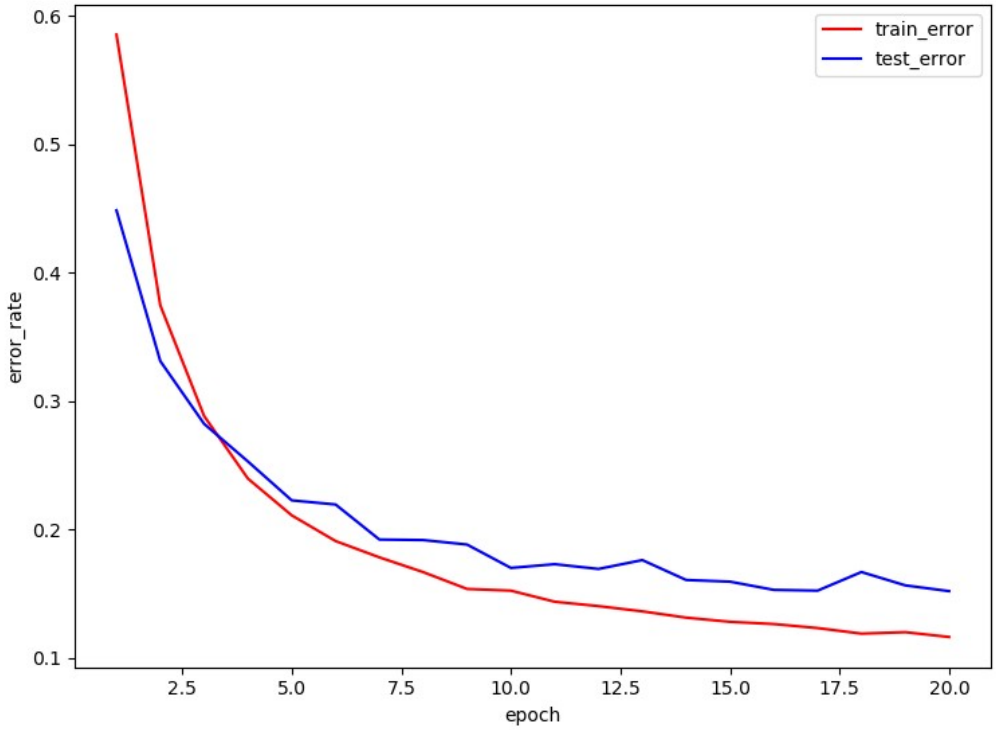
对比常规CNN，我们可以发现这里图中每隔两个卷积层之间会出现一个连接箭头，这个箭头官方称之为“shortcut”，我们翻译为“快捷链接”。我们将输入的图片作为x输入，经过第i个卷积层的输出设为x_i，例如，图中经过第一个卷积层的输出为x_1，第二个卷积层的输出为x_2...在x_1和x_3之间有一个shortcut。

一开始，图片作为输入x输入进第一个卷积层，第一个卷积层的输出为x_1，同时x_1也是第二个卷积层的输入。但是在第三和第四个卷积层之间，就不一样了。由于shortcut的存在，第四个卷积层的输入并不是第三个卷积层的输出，而是$ReLu(x_3+x_1)$。

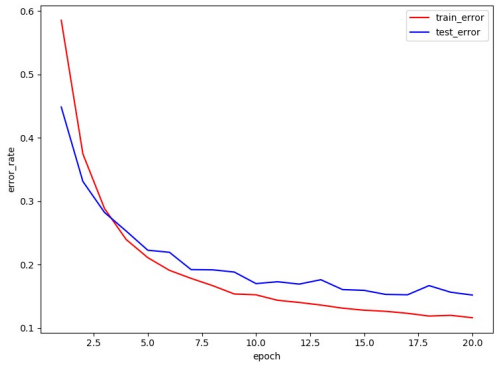
我们将每个shortcut包含的一部分称为“残差块”。“残差块”是ResNet网络的核心部分，也是ResNet网络分类准确率高，训练难度小的原因所在。完整的ResNet网络就是由若干个“残差块”串联而成，输入图片x经过若干个残差块后，再经过一个平均池化层、全连接层和softmax层后，便可得到我们想要的输出结果。

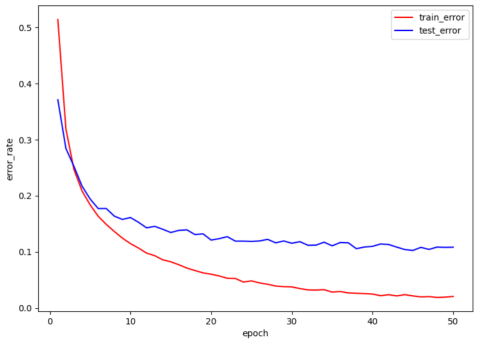
结果分析

我们首先设定epoch=20，学习率lr=0.01，batch_size=16，所得统计结果如下：

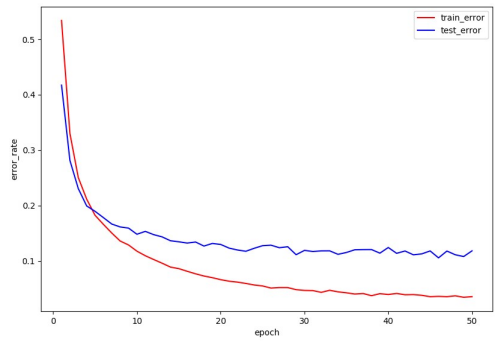


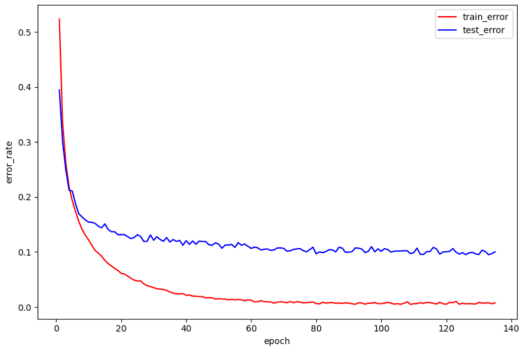
由图可知，最终训练集错误率在11%左右，测试集错误率在15%左右。我们尝试修改epoch、lr、batch_size来试着优化网络的效果：





p1: epoch=20 lr=0.01 batch_size=16　p2: epoch=50 lr=0.001 batch_size=16





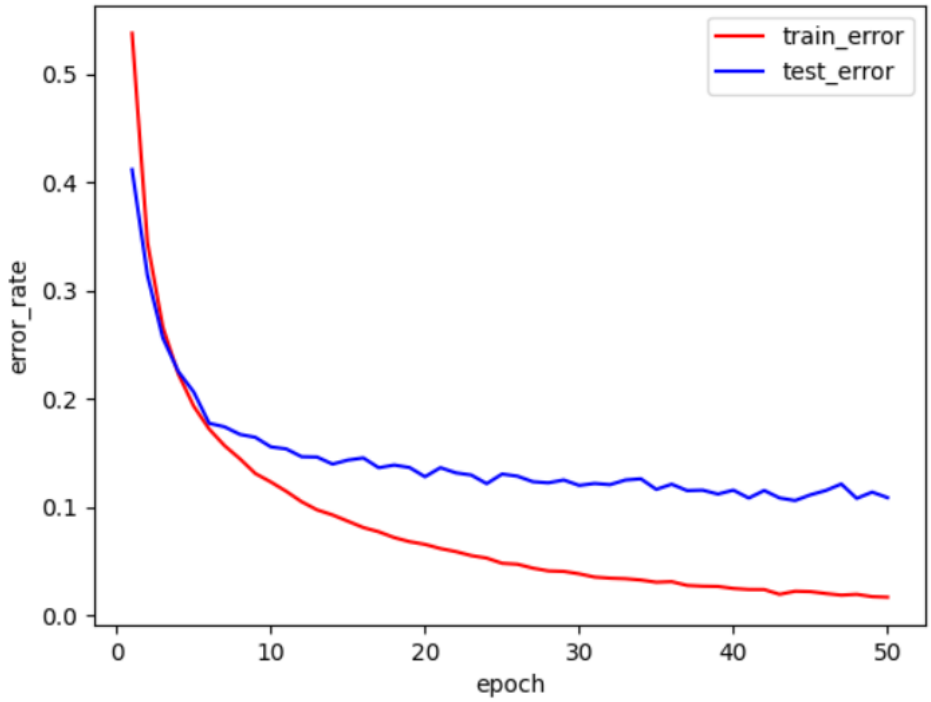
p3: epoch=50 lr=0.01 batch_size=64　p4: epoch=135 lr=0.001 batch_size=32

通过上面的对比，第4副图中的效果最好，最终的测试准确率在90%左右。将最终保存的模型带入测试接口中对测试集进行测试，结果如下：

```
cuda
测试样本总数：10000
ResNet分类正确个数：8916
ResNet分类错误个数：1084
ResNet分类准确率：0.891600
ResNet分类错误率：0.108400
损失函数值loss：0.026654

Process finished with exit code 0
```

其他尝试

我们尝试在ResNet18的基础上修改全连接层的个数，使其成为更为复杂的非线性关系，我们将ResNet18中的全连接层由1层增加到2层，全连接层的神经元个数为512-256-10。所得结果如下：




```
cuda
测试样本总数：10000
ResNet分类正确个数：8884
ResNet分类错误个数：1116
ResNet分类准确率：0.888400
ResNet分类错误率：0.111600
损失函数值loss：0.028911
```

对比上面的ResNet18我们可以发现，准确率并没有提高，因此，我们最终还是采用原来的一层全连接层，神经元个数为512。

实验中遇到的问题以及收获

tensor和Tensor的区别

在实验中，我们发现tensor和Tensor是两种不同的类。中途产生了Tensor和tensor之间进行转换的问题，后来通过查资料发现两者区别如下：

在PyTorch 中，`torch.Tensor`是一种主要的`tensor`类型，是`torch.FloatTensor()`的别名。所有的`tensor`都是`torch.Tensor`的实例。

而torch.tensor()是一个函数，函数原型是：

```
torch.tensor(data, dtype=None, device=None, requires_grad=False)
```

区别

`torch.Tensor(data)`：将输入的data转化`torch.FloatTensor()`

`torch.tensor(data)`：将data转化为`torch.FloatTensor`、`torch.LongTensor`、`torch.DoubleTensor`等类型，转化类型依据于`data`的类型或者`dtype`的值。

`torch.Tensor()`可以创建一个空的FloatTensor，使用`torch.tensor()`时则会报错。

内存问题

在写模型测试接口时，我们先加载模型，然后将数据输入。但在这个过程中跟训练模型不太相同。**训练模型时我们需要用到计算梯度和反向传播，但测试模型时我们不用。因此在测试接口中如果不添加相应步骤会导致GPU内存不足，而这大部分都是被没有清零的梯度占满的。**

例如：

```
# 在训练模型中我们往往会有这个语句
optimizer.zero_grad()  # 梯度清零
```

这一句不仅将梯度清零，还释放了GPU内存，因此训练模型过程中不会出现GPU内存不足的情况。

而在测试接口中，我们一般不会用到optimizer这个库，因此，导入网络和数据测试时不会有梯度清零和释放内存的步骤，会出现如下报错：

为了避免这种情况，在测试接口中我们需要添加`with torch.no_grad():`这一句，表面**数据不需要梯度计算，即不会进行反相传播**，因此不会占用内存。

以ResNet_port.py的测试接口为例：

```
for i, (inputs, labels) in enumerate(test_loader):
    with torch.no_grad():  # 测试网络时不需要梯度，使用这句防止内存被占用导致无法运行
        inputs, labels = Variable(inputs), Variable(labels)  # 转化为变量，里面的值会随时改变，用于迭代
        inputs, labels = inputs.to(device), labels.to(device)  # 将数据放入GPU运算

        # 记录ResNet错误个数
        ResNet_outputs = ResNet(inputs)
        ResNet_outputs_list = ResNet_outputs.data.tolist()
        ResNet_outputs_labels = []
        ResNet_loss += criterion(ResNet_outputs, labels)  # 计算损失值
        for j in range(batch_size):  # 逐行找分类结果
            if j >= len(ResNet_outputs_list):
                break  # 数据集不能杯batch_size整除
            result = ResNet_outputs_list[j].index(max(ResNet_outputs_list[j]))
            ResNet_outputs_labels.append(result)  # 找出数值最大所在的标签
            if ResNet_outputs_labels[j] != labels.data.tolist()[j]:
                ResNet_error += 1  # 记录错误个数
```

这样就可以防止梯度占用内存的情况了。

总结

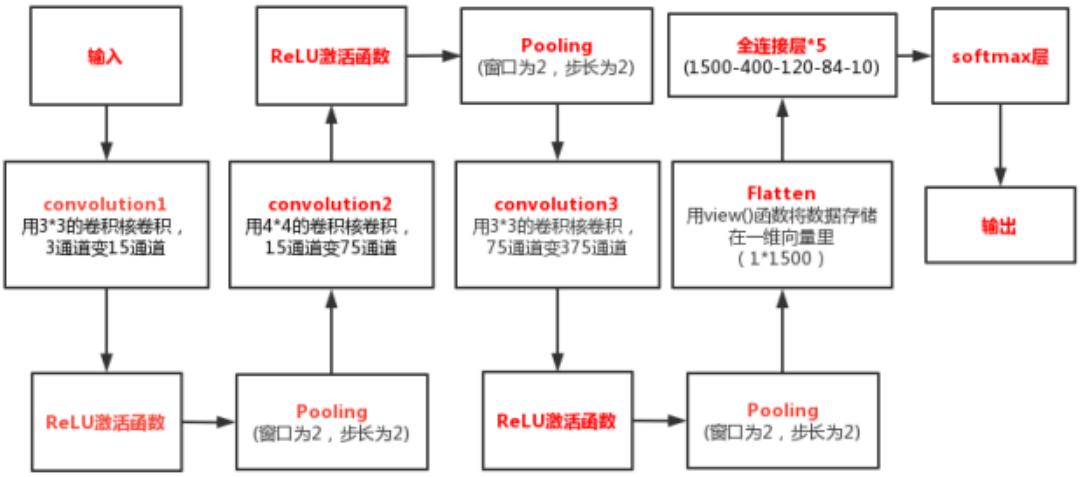
本次实验中，我们采用了常规CNN和ResNet18来对cifar10进行分类。考虑到网络的轻量化，并没有使用ResNet50甚至ResNet101。

对于最终的训练结果，我们拟以下表格：

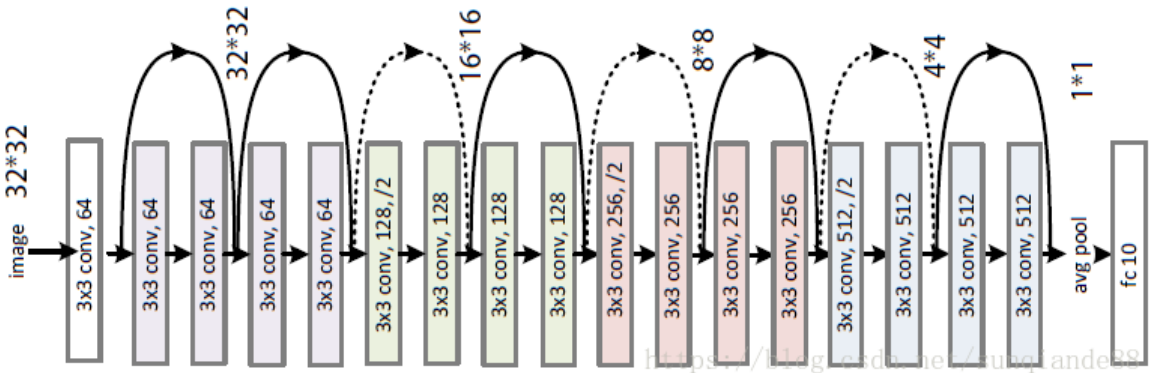
网络模型	训练集错误率	测试集错误率
CNN	5.312%	19.84%
ResNet18	1.134%	10.84%

作为轻量级网络，我们搭建的CNN和ResNet18都有较为优秀的准确率。老师和助教只需在CNN_port.py和ResNet_port.py中修改数据集路径即可导入已训练好的模型对测试集进行测试。

最终采用的CNN结构如下：



最终采用的ResNet结构如下：



小组分工

郭凯 20354034　CNN、ResNet18网络搭建，代码编写，CNN、ResNet网络训练，写报告　评分：10

王浩祯20354248　CNN网络训练，调整参数　评分：9

曾陆豪 20354192　CNN、ResNet网络训练，写报告　评分：9

陈清桦20354014　写报告，调整网络参数　评分：9

陈柔柔20354015　写报告，调整网络参数　　评分：9

陈光燕20354007　写报告，调整网络参数　　评分：9