人工智能作业2

1852824 吴杨婉婷

- 人工智能作业2
  - 1. 作业需求描述
  - 2. Requirements
  - 3. 模型代码详解
    - 3.1 ResNet提出
    - 3.2 ResNet解决网络退化问题
    - 3.3 ResNet实现
    - 3.5 VGG原理
    - 3.6 VGG16实现
    - 3.7 VGG优缺点
  - 4.实验结果
    - 4.1 简单cnn
    - 4.2 数据增强
    - 4.3 正则化
    - 4.4 resnet
    - 4.5 resnet微调
    - 4.6 VGG微调
  - 5.结果分析与心得体会
    - 5.1 数据增强
    - 5.2 l2正则化
    - 5.3 微调
    - 5.4 总结
  - 6.主体代码解释
  - 7.作者

1. 作业需求描述

1. 使用简简单CNN作为分类器完成Cifar10分类任务
2. 使用简单CNN作为分类器，增加数据增强、正则化等技巧，完成Cifar10分类任务
3. 使用ResNet作为分类器，完成Cifar10分类任务
4. 使用VGG或ResNet预训练模型，微调后作为分类器，完成Cifar10分类任务

2. Requirements

- **Development Environment:**

  Win 10

- **Development Software:**

  PyCharm *2020.3.5.PC-191.6605.12*

- **Development Language:**

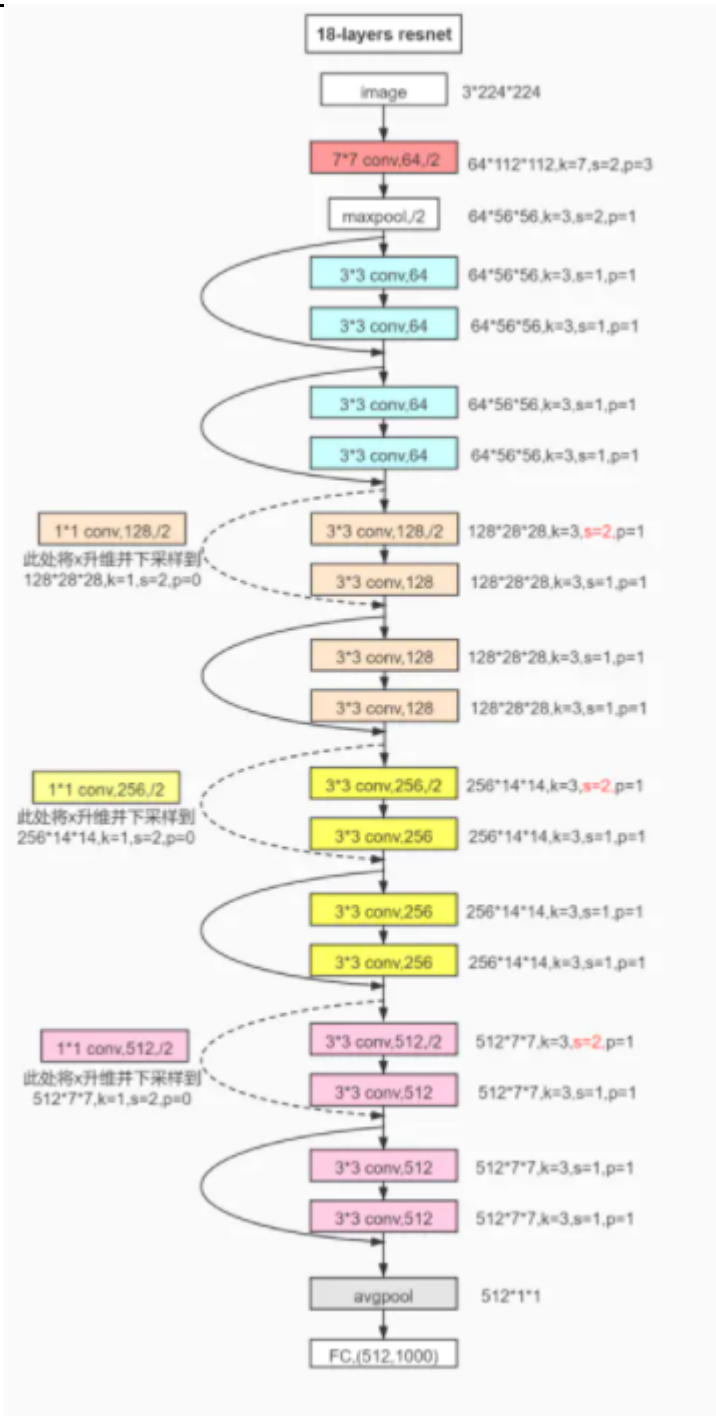
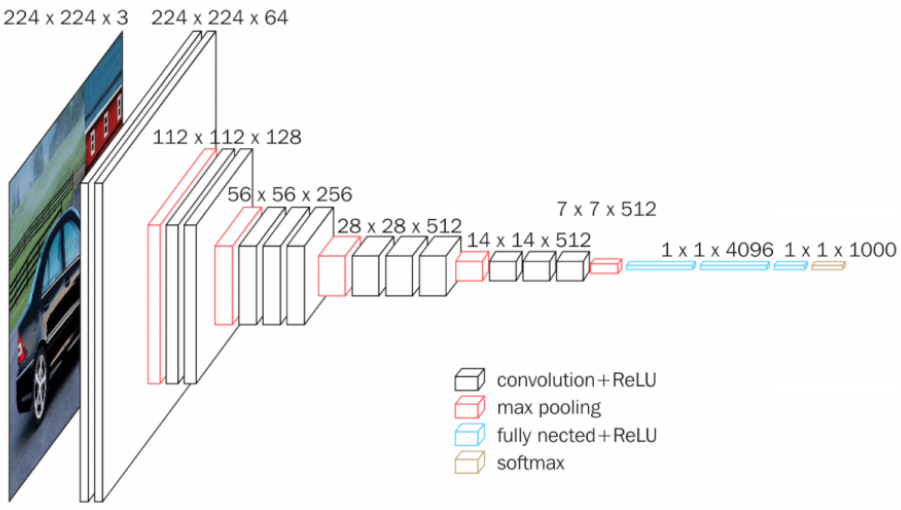
  Python

- **Mainly Reference Count:**

1. torchvision
2. matplotlib
3. os
4. torch
5. numpy

3. 模型代码详解

number	描述	model
1	简单cnn先用6个5* 5的卷积核，然后进行池化，再用16个5*5的卷积核，最后经过三层全连接	Simple_CNN_Net
2		ResNet18

number	描述	model
		
3		VGG16

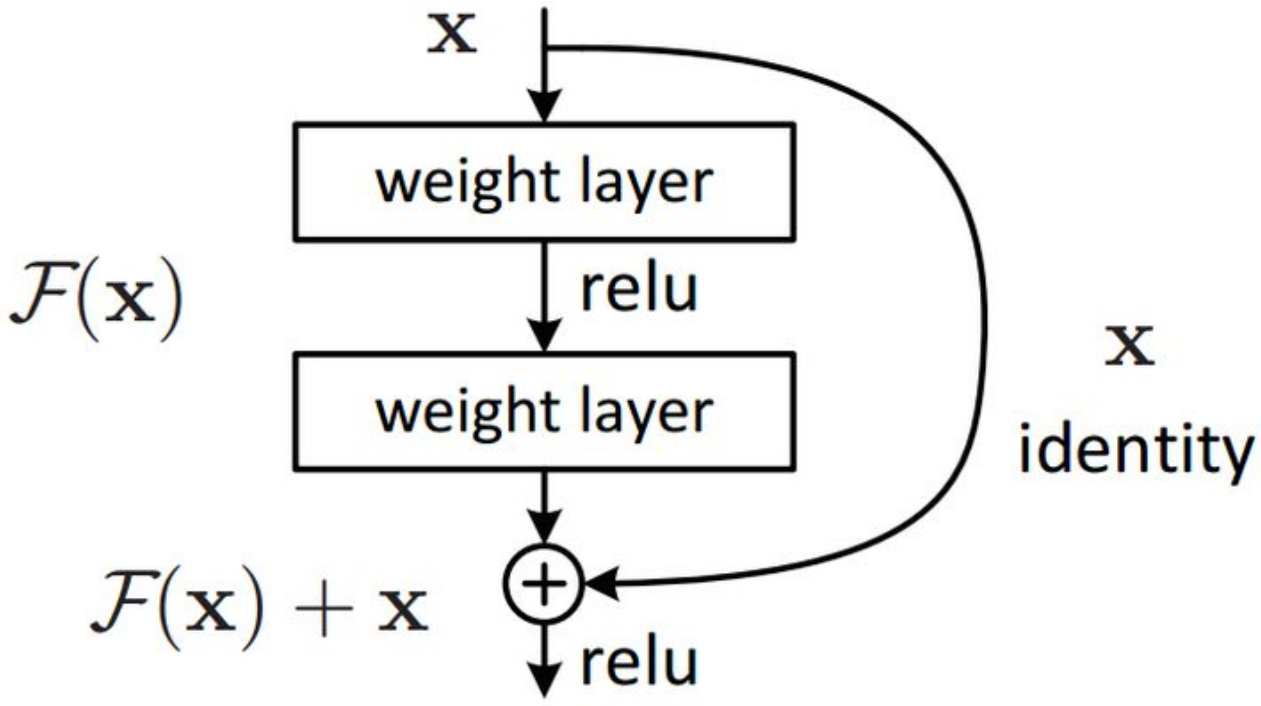
3.1 ResNet提出

- 自从深度神经网络在ImageNet大放异彩之后，后来问世的深度神经网络就朝着网络层数越来越深的方向发展。直觉上我们不难得出结论：增加网络深度后，网络可以进行更加复杂的特征提取，因此更深的模型可以取得更好的结果。但事实并非如此，人们发现随着网络深度的增加，模型精度并不总是提升，并且这个问题显然不是由过拟合（overfitting）造成的，因为网络加深后不仅测试误差变高了，它的训练误差竟然也变高了。作者提出，这可能是因为更深的网络会伴随梯度消失/爆炸问题，从而阻碍网络的收敛。作者将这种加深网络深度但网络性能却下降的现象称为退化问题（degradation problem）。
- 当传统神经网络的层数从20增加为56时，网络的训练误差和测试误差均出现了明显的增长，也就是说，网络的性能随着深度的增加出现了明显的退化。ResNet就是为了解决这种退化问题而诞生的。

3.2 ResNet解决网络退化问题

- 随着网络层数的增加，梯度爆炸和梯度消失问题严重制约了神经网络的性能，研究人员通过提出包括Batch normalization在内的方法，已经一定程度上缓解了这个问题，但依然不足以满足需求。

- 问题解决的标志是：增加网络层数，但训练误差不增加。20层的网络是56层网络的一个子集，56层网络的解空间包含着20层网络的解空间。如果我们将56层网络的最后36层全部短接，这些层进来是什么出来也是什么,也就是做一个恒等映射.因为梯度消失现象使得网络难以训练，虽然网络的深度加深了，但是实际上无法有效训练网络，训练不充分的网络不但无法提升性能，甚至降低了性能。



3.3 ResNet实现

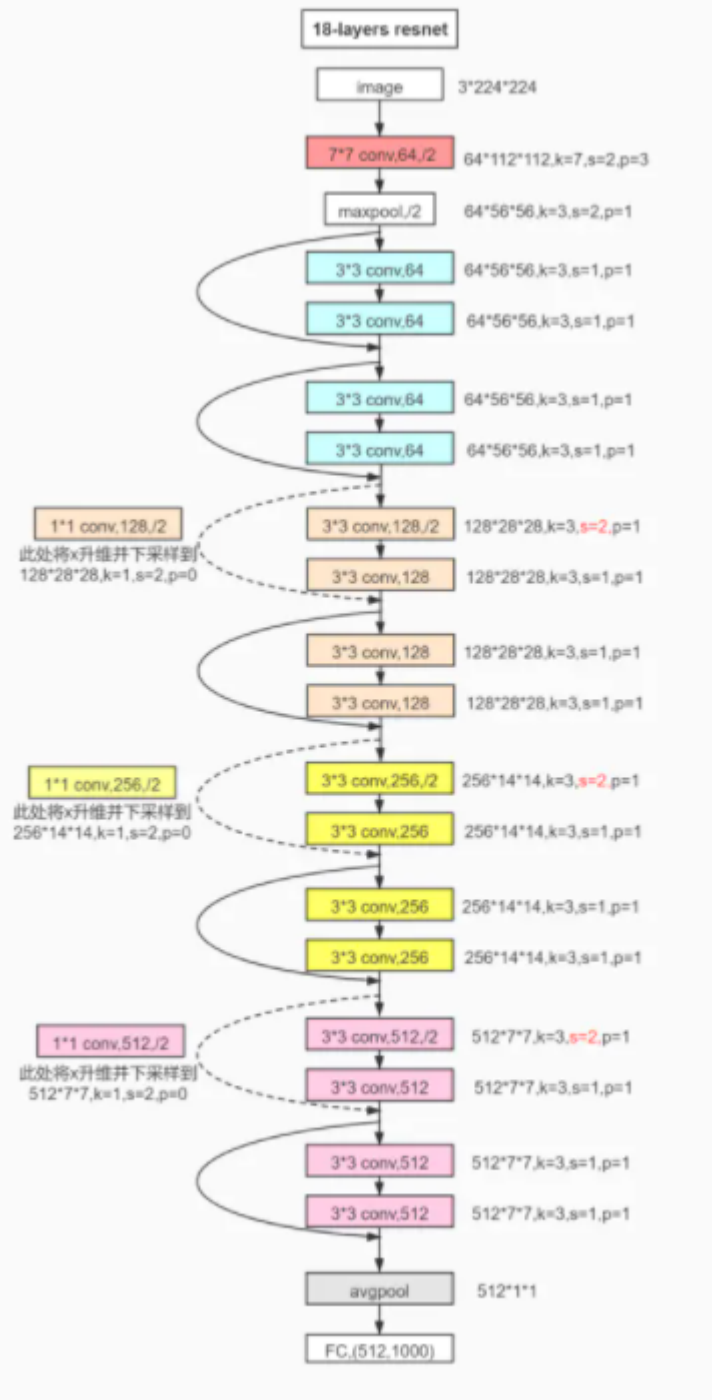
- 在ResNet类中的forward( )函数规定了网络数据的流向：
    1. 数据进入网络后先经过输入部分（conv1, bn1, relu, maxpool）；

2. 然后进入中间卷积部分（layer1, layer2, layer3, layer4）；

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2 $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

3. 最后数据经过一个平均池化和全连接层（avgpool, fc）输出得到结果；



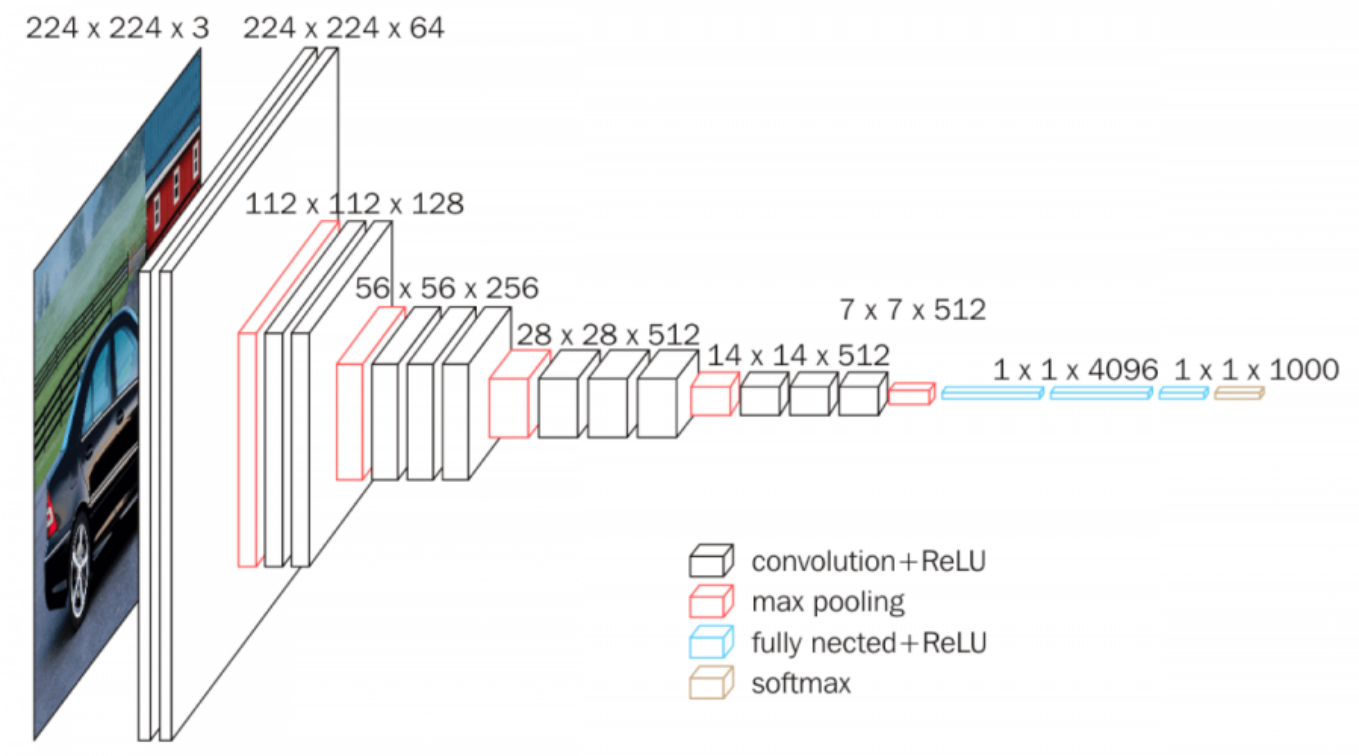
4.

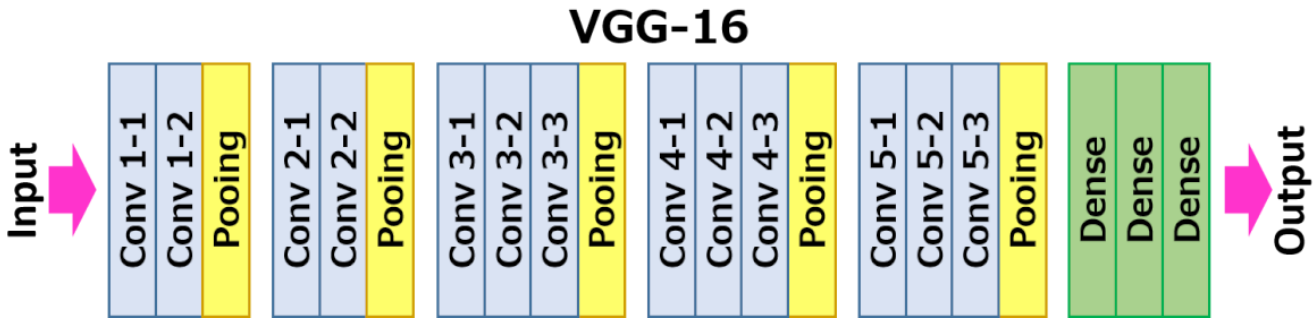
3.5 VGG原理

- VGG16包含了16个隐藏层（13个卷积层和3个全连接层）
- VGG16一个改进是采用连续的几个3x3的卷积核代替AlexNet中的较大卷积核（11x11，7x7，5x5）。对于给定的与输出有关的输入图片的局部大小，采用堆积的小卷积核是优于采用大的卷积核，因为多层非线性层可以增加网络深度来保证学习更复杂的模式，而且代价还比较小（参数更少）。在VGG中，使用了3个3x3卷积核来代替7x7卷积核，使用了2个3x3卷积核来代替5*5卷积核，这样做的主要目的是在保证具有相同感知野的条件下，提升了网络的深度，在一定程度上提升了神经网络的效果。比如，3个步长为1的3x3卷积核的一层层叠加作用可看成一个大小为7的感受野（其实就表示3个3x3连续卷积相当于一个7x7卷积），其参数总量为 3x(9xC^2)，如果直接使用7x7卷积核，其参数总量为 49xC^2，这里 C 指的是输入和输出的通道数。很明显，27xC^2小于49xC^2，即减少了参数；而且3x3卷积核有利于更好地保持图像性质。

3.6 VGG16实现

- VGG16包含了16个隐藏层（13个卷积层和3个全连接层）





3.7 VGG优缺点

- VGG优点

  VGGNet的结构非常简洁，整个网络都使用了同样大小的卷积核尺寸（3x3）和最大池化尺寸（2x2）。几个小滤波器（3x3）卷积层的组合比一个大滤波器（5x5或7x7）卷积层好：验证了通过不断加深网络结构

可以提升性能。

- VGG缺点

  VGG耗费更多计算资源，并且使用了更多的参数（这里不是3x3卷积的），导致更多的内存占用（128M）。其中绝大多数的参数都是来自于第一个全连接层。

```
class Simple_CNN_Net(nn.Module):
    def __init__(self):
        super(Simple_CNN_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
```

```
class ResidualBlock(nn.Module):
    def __init__(self, inchannel, outchannel, stride=1):
        super(ResidualBlock, self).__init__()
        self.left = nn.Sequential(
            nn.Conv2d(inchannel, outchannel, kernel_size=3, stride=stride,
padding=1, bias=False),
            nn.BatchNorm2d(outchannel),
            nn.ReLU(inplace=True),
            nn.Conv2d(outchannel, outchannel, kernel_size=3, stride=1, padding=1,
bias=False),
            nn.BatchNorm2d(outchannel)
        )
        self.shortcut = nn.Sequential()
        if stride != 1 or inchannel != outchannel:
            self.shortcut = nn.Sequential(
                nn.Conv2d(inchannel, outchannel, kernel_size=1, stride=stride,
bias=False),
                nn.BatchNorm2d(outchannel)
            )

    def forward(self, x):
        out = self.left(x)
        out += self.shortcut(x)
        out = F.relu(out)
```

```
        return out

class ResNet(nn.Module):
    def __init__(self, ResidualBlock, num_classes=10):
        super(ResNet, self).__init__()
        self.inchannel = 64
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),
        )
        self.layer1 = self.make_layer(ResidualBlock, 64,  2, stride=1)
        self.layer2 = self.make_layer(ResidualBlock, 128, 2, stride=2)
        self.layer3 = self.make_layer(ResidualBlock, 256, 2, stride=2)
        self.layer4 = self.make_layer(ResidualBlock, 512, 2, stride=2)
        self.fc = nn.Linear(512, num_classes)

    def make_layer(self, block, channels, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)   #strides=[1,1]
        layers = []
        for stride in strides:
            layers.append(block(self.inchannel, channels, stride))
            self.inchannel = channels
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv1(x)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out


def ResNet18():

    return ResNet(ResidualBlock)
```

```
class VGG16(nn.Module):
    def __init__(self, num_classes=10):
        super(VGG16, self).__init__()
        self.features = nn.Sequential(
            # 1
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            # 2
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
```

```
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 3
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            # 4
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 5
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            # 6
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            # 7
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 8
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            # 9
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            # 10
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 11
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            # 12
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            # 13
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.AvgPool2d(kernel_size=1, stride=1),
        )
        self.classifier = nn.Sequential(
```

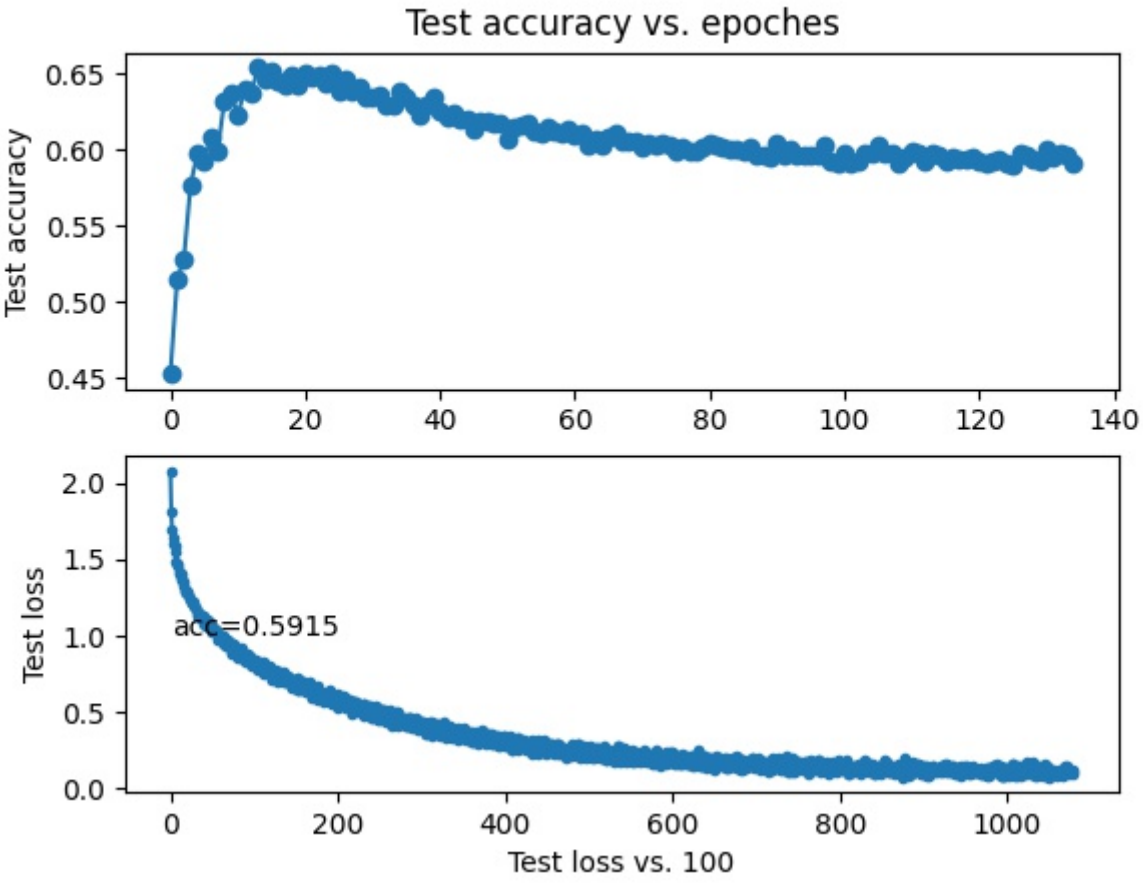
```
            # 14
            nn.Linear(512, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            # 15
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            # 16
            nn.Linear(4096, num_classes),
        )
        # self.classifier = nn.Linear(512, 10)

    def forward(self, x):
        out = self.features(x)
        #         print(out.shape)
        out = out.view(out.size(0), -1)
        #         print(out.shape)
        out = self.classifier(out)
        #         print(out.shape)
        return out
```

4.实验结果

number	model	acc
1	简单cnn	0.6415
2	数据增强	0.6619
3	正则化	0.6762
4	resnet	0.8392
5	resnet微调	0.9126
6	VGG微调	0.9012

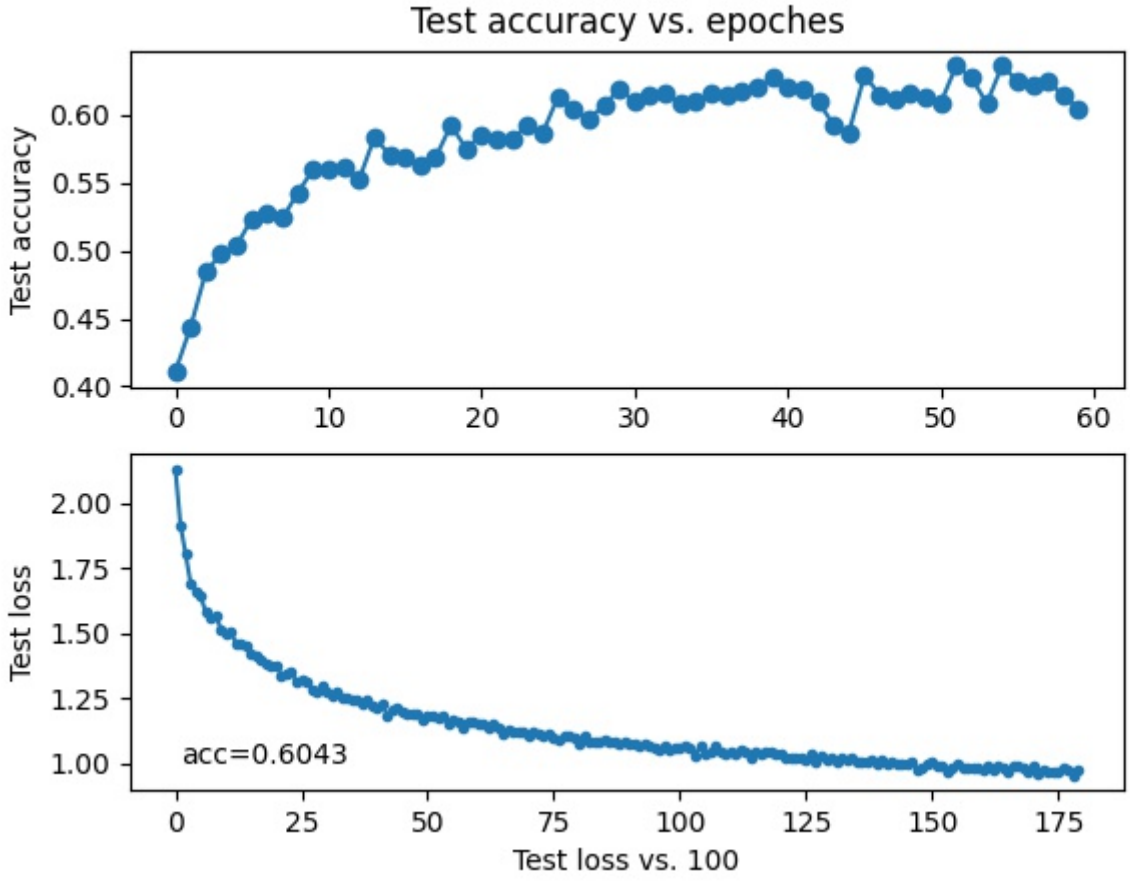
4.1 简单cnn



4.2 数据增强

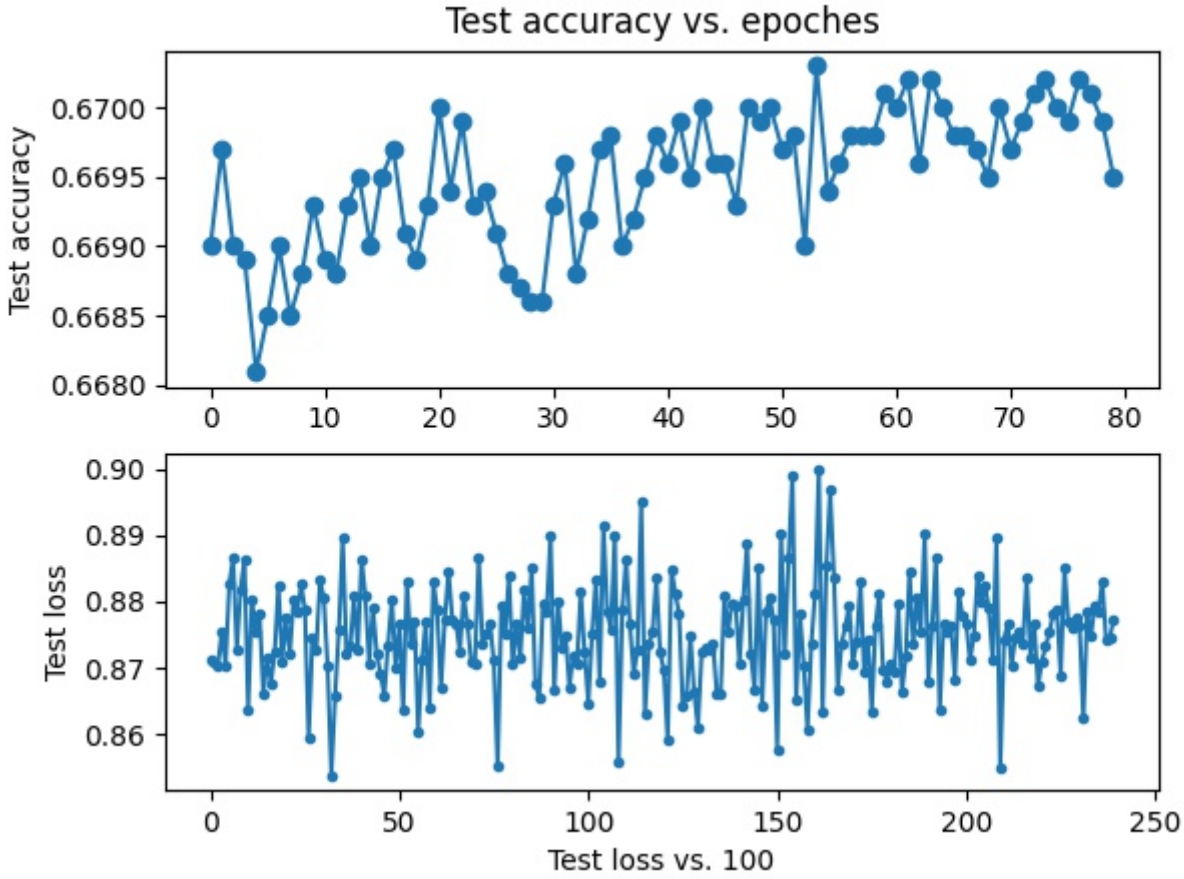
- 核心代码
- 将图片进行Resize，RandomHorizontalFlip，RandomCrop和归一化

```
train_transform = transforms.Compose([transforms.Resize(40),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.RandomCrop(32),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.5, 0.5, 0.5],
[0.5, 0.5, 0.5])])
test_transform = transforms.Compose([transforms.ToTensor(),
                                     transforms.Normalize([0.5, 0.5, 0.5],
[0.5, 0.5, 0.5])])
```

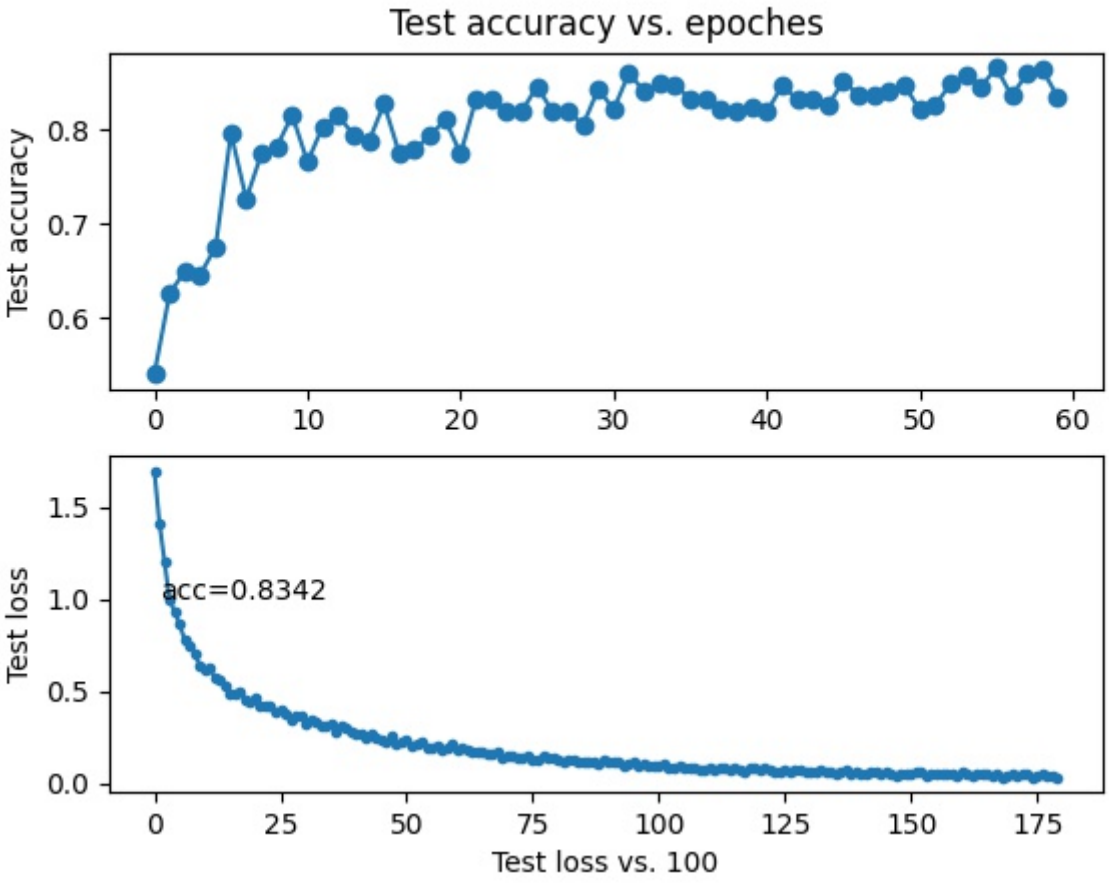


4.3 正则化

- 为了训练方便，我使用了之前训练好的数据数据，作图的坐标轴不是从0开始，所以在图中会感觉有震荡
- 采用LR = 0.0007，L2正则化，同时根据之前训练的结果调整了学习率在0.00001

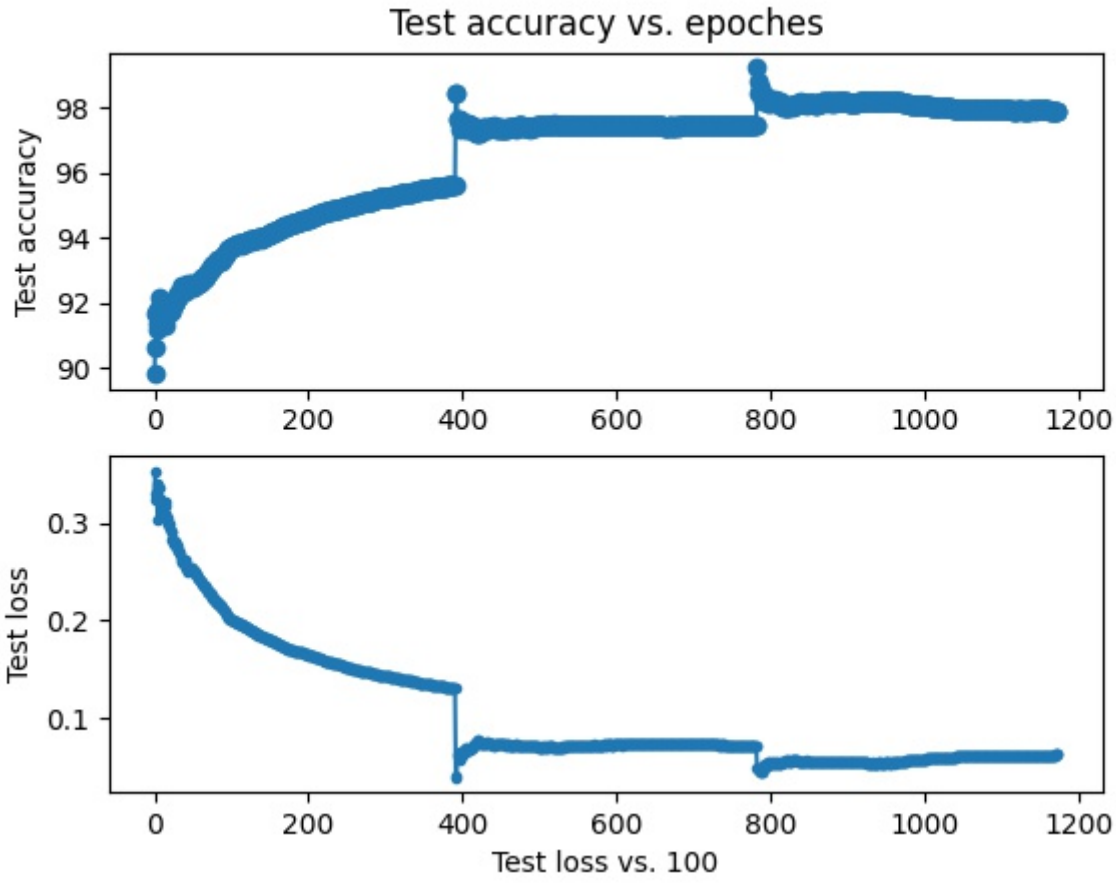

Test accuracy vs. epoches
Test accuracy
0.6700
0.6695
0.6690
0.6685
0.6680
0
10
20
30
40
50
60
70
80
Test loss
0.90
0.89
0.88
0.87
0.86
0
50
100
150
200
250
Test loss vs. 100

4.4 resnet



4.5 resnet微调

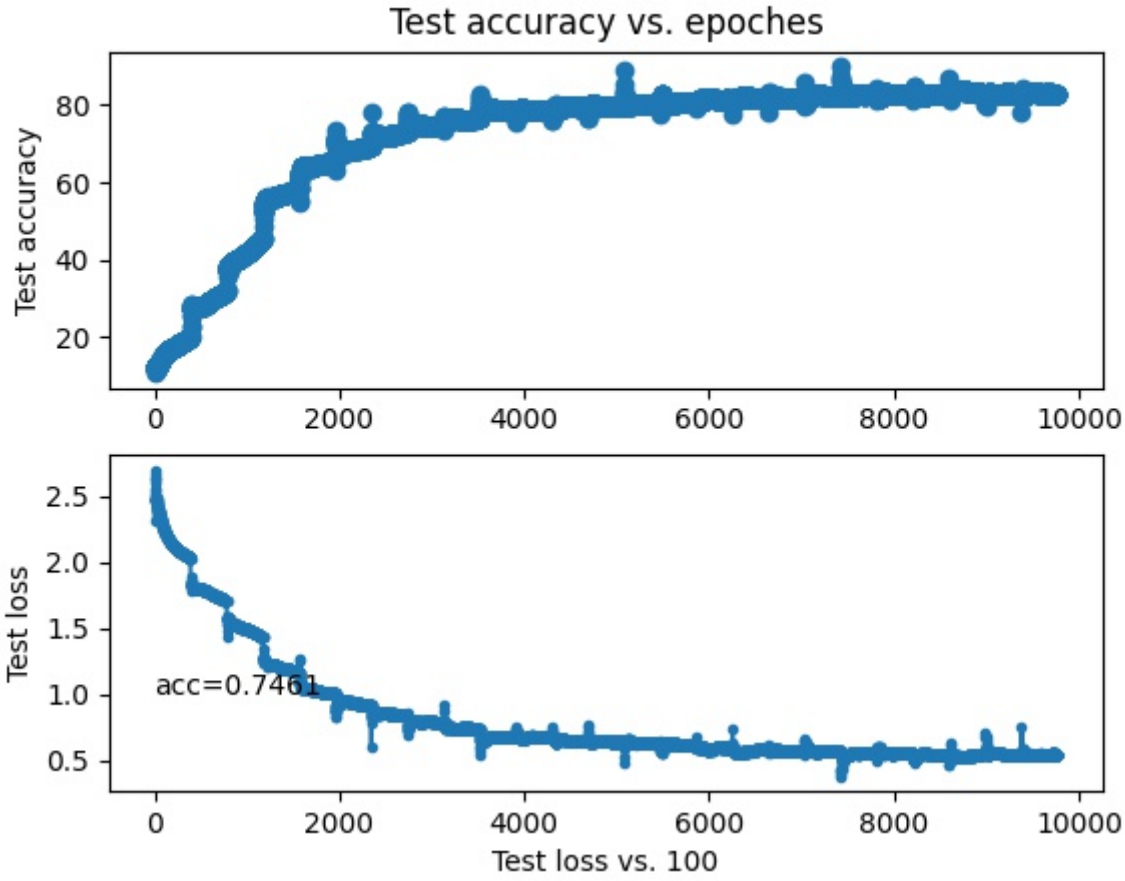
- 首先是随着iteration的改变调整学习率，从0.1降到0.001
- 优化方式为mini-batch momentum-SGD，并采用L2正则化（权重衰减）LR = 0.0007



4.6 VGG微调

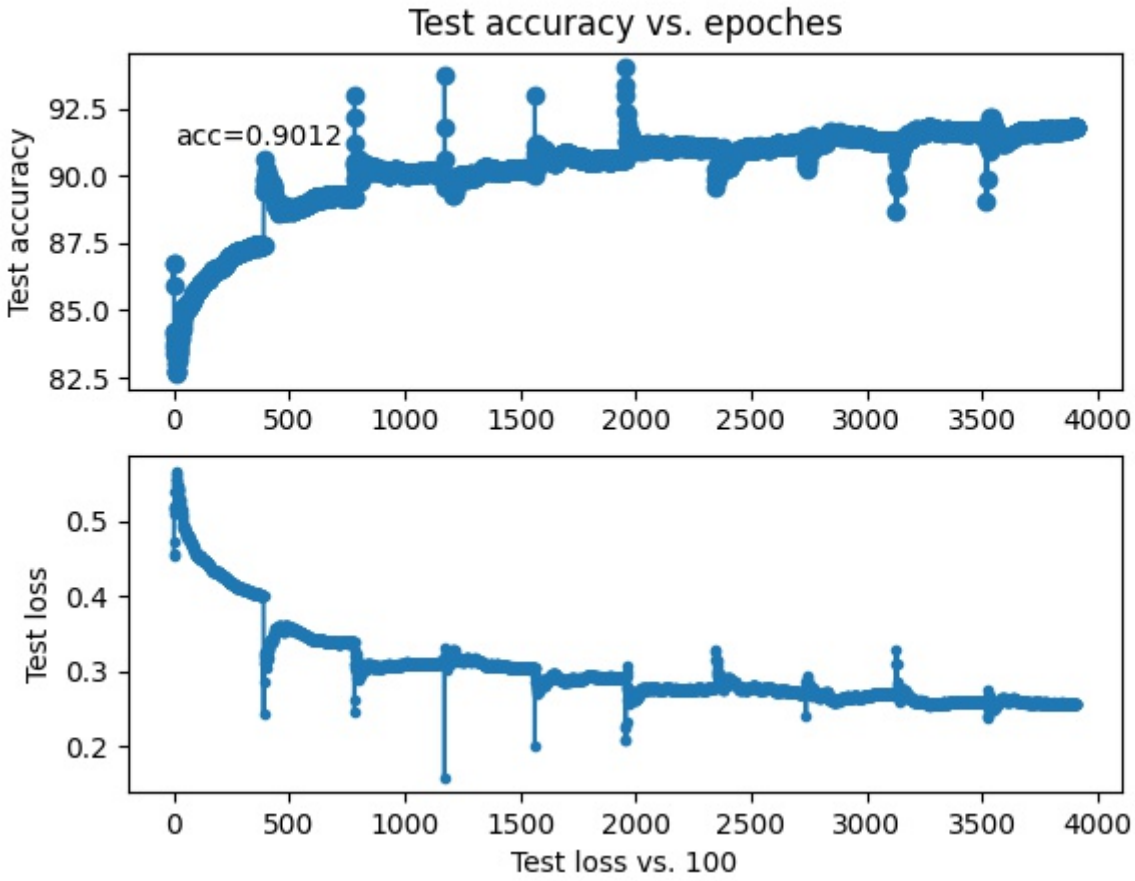
- 首先是随着iteration的改变调整学习率，从0.1降到0.001
- 优化方式为mini-batch momentum-SGD，并采用L2正则化（权重衰减）LR = 0.0007

从零开始调整



用预

先训练好的参数微调



5.结果分析与心得体会

5.1 数据增强

- 使用简单的cnn网络来作为cifar10的分类器，最好的准确率在 0.6415 ，加入数据增强之后，准确率到达0.6619，说明数据增强在图像领域是比较有效的

- 首先增强的样本和原来的样本是由强相关性的（裁剪、翻转、旋转、缩放、扭曲等几何变换，还有像素扰动、添加噪声、光照调节、对比度调节、样本加和或插值、分割补丁等）。即我们强制网络学习了某些样本变换方式，而如果这些变换方式使得网络的性能有所提升，那么，可以简单的认为网络在之前并没有学到相关的变换，或者学的并不全面。而如果通过某些简单的操作，提高了最终性能，这说明网络可能并没有我们想象的那种方式去拟合数据，比如简单的平移不变性。如果我们通过简单的裁剪、平移提高了模型的性能，那么，恰恰说明网络可能并没有完全学习到平移不变性。数据增强可以带来某种正则化作用的，这样就可以减小模型的结构风险。

- 我只是使用了增强的数据来训练模型，并没有用增强后的数据来测试模型。对于改进有一个猜想，这里可以用一个数据增强、非增强，分别在训练和测试时做一个排列组合，一共四种情况。可以测试出数据增强对网络性能带来的增益，以及引入的经验风险（有些数据增强方法产生的图像可能并不出现在现实场景中），当然还可以测试出网络是否真的学会了某种视觉不变性。

5.2 l2正则化

- L2正则化在原先的损失函数后边再加多一项，那加上L2正则项的损失函数就可以表示为：$L(\theta)=L(\theta)+\lambda\sum ni\theta 2i$，其中θ就是网络层的待学习的参数，λ则控制正则项的大小，较大的取值将较大程度约束模型复杂度，反之亦然。L2约束通常对稀疏的有尖峰的权重向量施加大的惩罚，而偏好于均匀的参数。这样的效果是鼓励神经单元利用上层的所有输入，而不是部分输入。所以L2正则项加入之后，权重的绝对值大小就会整体倾向于减少，尤其不会出现特别大的值（比如噪声），即网络偏向于学习比较小的权重。
- 通过引入正则化，使模型参数偏好比较小的值，有针对的减小了模型容量。大的参数值对应于波动剧烈的函数，小的参数值对应于比较平缓的参数，发生过拟合时的函数波动会比较大，过拟合的模型往往具有比较大的参数，因此正则化将这部分模型族从假设空间中剔除掉，发生过拟合的可能就变小
- 加入L2正则化后，简单cnn网络准确度从0.6619增加到0.6762

5.3 微调

- 我认为最有用的微调是根据训练数据准确率，loss,方差等调整学习率，考虑引入早衰，在loss变化的特别小甚至因为过拟合准确率下降的时候提前结束训练
- 采用optim.SGD优化器
- 这里有个疑惑的地方，在准确率低于90时不断调整学习率对准确度的提高很有帮助，但最后的准确率达到0.910左右的时候无论如何调整参数都没有办法再提高学习率了，不知道是否是陷入了局部最优解（因为查阅资料发现resnet18在cifar10上的准确率为95，vgg16在cifar10上的准确率为94）

5.4 总结

- 学习了许多不同的调参技巧，简单了解原理
- 因为在colab云计算，没有在自己的电脑上运行代码，对训练速度和显卡利用率等方面的优化没有做深入的了解，没有充分利用显卡资源，但是因为是云环境，对断点保存数据，恢复现场积累了一点点经验
- 总体来说感觉数据集大小十分关键，再其次就是根据训练情况调整学习率大小，考虑引入早衰或者考虑动态调整学习率大小，至于数据增强，优化器，和正则化是调参必备

- 对比MNIST和Fashion-MNIST的图像识别情况，我认为主要原因之一是Cifar10图像为RGB有三个通道，且像素点明显多于MNIST和Fashion-MNIST，数据量也更大，模型运行时间长、准确率还像mnist一样达到99%一样不大现实

6.主体代码解释

- 此处仅以resnet18微调为例解释代码，vgg16的微调和resnet18主体代码完全一致
- 数据集下载

```
# 准备数据集并预处理
    train_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((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
# R,G,B每层的归一化用到的均值和方差
    ])

    test_transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
    ])

    trainset = datasets.CIFAR10(root='./data', train=True, download=False,
                                            transform=train_transform)  # 训练数据集
    train_loader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE, shuffle=True,
                                               num_workers=2)  # 生成一个个batch进行批训练，组成batch的时候顺序打乱取

    testset = datasets.CIFAR10(root='./data', train=False, download=False, transform=test_transform)
    test_loader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, num_workers=2)
```

- 模型定义

```
# 模型定义-ResNet
    net = mymodel.ResNet18().to(device)
    # 取得之前的参数
    # net.load_state_dict(torch.load('./parameterForCifar10/parameter5.pkl'))

    # 定义损失函数和优化方式
    criterion = nn.CrossEntropyLoss()  # 损失函数为交叉熵，多用于多分类问题
    optimizer = optim.SGD(net.parameters(), lr=LR, momentum=0.9,
```

```
                            weight_decay=5e-4)  # 优化方式为mini-batch momentum-SGD,
并采用L2正则化（权重衰减）
```

- 训练开始

```
# 训练
    Loss_list = []
    Accuracy_list = []
    for epoch in range(pre_epoch, EPOCH):
        net.train()
        sum_loss = 0.0
        correct = 0.0
        total = 0.0
        for i, data in enumerate(train_loader, 0):
            length = len(train_loader)
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()

            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()


        # 每训练完一个epoch测试一下准确率
        print("Waiting Test!")
        with torch.no_grad():
            correct = 0
            total = 0
            for data in test_loader:
                net.eval()
                images, labels = data
                images, labels = images.to(device), labels.to(device)
                outputs = net(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum()
```

- 开始测试

```
  net.eval()  # 将模型变换为测试模式
    correct = 0
    total = 0
    for data_test in test_loader:
        images, labels = data_test
        output_test = net(images)
```

```
        _, predicted = torch.max(output_test, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum()
    print("correct: ", correct)
    print("Test acc: {0}".format(correct.item() / len(test_loader.dataset)))
```

- 保存参数

```
 # 保存
    if not os.path.exists("./parameterForCifar10"):
        os.mkdir("./parameterForCifar10")
    torch.save(net.state_dict(),
'./parameterForCifar10/parameter{}.pkl'.format(choice))
```

- 作图

```
 # 作图
    x1 = range(0, len(Accuracy_list))
    x2 = range(0, len(Loss_list))
    y1 = Accuracy_list
    y2 = Loss_list
    plt.subplot(2, 1, 1)
    plt.plot(x1, y1, 'o-')
    plt.title('Test accuracy vs. epoches')
    plt.ylabel('Test accuracy')
    plt.subplot(2, 1, 2)
    plt.plot(x2, y2, '.-')
    plt.xlabel('Test loss vs. 100')
    plt.ylabel('Test loss')
    plt.text(x=1,y=1,s="acc={}".format(((correct.item() /
len(test_loader.dataset)) )))

plt.savefig("./parameterForCifar10/Cifar10_accuracy_loss{0}.jpg".format(choice))
    plt.show()
```

7.作者

ID	Name
1852824	吴杨婉婷

指导老师 唐堂老师

联系方式 email: 1852824@tongji.edu.cn