



**研究生课程考核试卷**

（适用于课程论文、提交报告）

科 目： 人工智能与深度学习导论 教 师： 冯涛

姓 名： 成永平 学 号： 201902110512

专 业： 计算机应用技术 类 别：（学术🗹、专业□）

上课时间：2020 年 3 月 至 2020 年 6月

考生成绩：

|  |  |  |
| --- | --- | --- |
| 卷面成绩 | 平时成绩 | 课程综合成绩 |
|  |  |  |

阅卷评语：

阅卷教师（签名）

研究生部制

摘 要

文章实现了基于PyTorch使用LENET5、VGG16和RESNET实现Cifar10的分类。介绍了人工智能和深度学习的发展历程和研究意义，同时也做多个主流框架的对比，选择PyTorch作为主体框架实现实验。Cifar10数据集是机器学习常用的数据集之一，同时LENET5、VGG16和RESNET作为CNN的延申，继承了CNN基本框架，也改进神经网络的效率，扩大了神经网络的使用。最后在云服务器上实现两种模型下不同数据集之间的对比，通过交叉熵损失函数LOSS和训练准确率ACC来评价模型的优劣。

关键字：分类；LENET5；VGG16; RESNET

目 录

[1研究背景与意义 1](#_Toc44438242)

[1.1研究背景 1](#_Toc44438243)

[1.2 研究意义 1](#_Toc44438244)

[2 数据集 3](#_Toc44438245)

[3 研究模型 4](#_Toc44438246)

[3.1 CNN的组成 4](#_Toc44438247)

[3.2 LENET5 5](#_Toc44438248)

[3.3 VGG16模型 7](#_Toc44438249)

[3.3.1 VGG 8](#_Toc44438250)

[3.3.2 VGG16结构 9](#_Toc44438251)

[3.4 RESNET 10](#_Toc44438252)

[4实验结果 13](#_Toc44438253)

[4.1 实验环境 13](#_Toc44438254)

[4.2 实验过程 13](#_Toc44438255)

[4.2.1 LENET5实现CIFAR10分类 13](#_Toc44438256)

[4.2.2 VGG16实现CIFAR10分类 15](#_Toc44438257)

[4.2.3 RESNET实现CIFAR10分类 18](#_Toc44438258)

[4.3 结果对比 20](#_Toc44438259)

[5总 结 22](#_Toc44438260)

[参考文献 23](#_Toc44438261)

[附录 24](#_Toc44438262)

基于PyTorch使用LENET5、VGG16和RESNET实现Cifar10的分类

# 1研究背景与意义

## 1.1研究背景

人工智能(Artificial Intelligence)，英文缩写为AI。它是研究、开发用于模拟、延伸和扩展人的智能的理论、方法、技术及应用系统的一门新的技术科学。人工智能是计算机科学的一个分支，它企图了解智能的实质，并生产出一种新的能以人类智能相似的方式做出反应的智能机器，该领域的研究包括语音识别、图像识别、机器人、自然语言处理、智能搜索和专家系统等。人工智能可以对人的意识、思维的信息过程的模拟 。人工智能不是人的智能，但能像人那样思考、也有可能超过人的智能。

深度学习(Deep Learning)的概念源于人工神经网络的研究。含多隐层的多层感知器就是一种深度学习结构。深度学习通过组合低层特征形成更加抽象的高层表示属性类别或特征，以发现数据的分布式特征表示。深度学习是机器学习研究中的一个新的领域，其动机在于建立、模拟人脑进行分析学习的神经网络，它模仿人脑的机制来解释数据，例如图像，声音和文本。

近些年来，随着计算机视觉计算的不断迭代更新，计算机的硬件如CPU，GPU和FPGA等计算能力的提升，图像识别的模型训练速度得到提升，由此，识别的准确率也有了很大的提升。在国外，手机移动端使用摄像头皮肤癌症检测对于采集的图像进行分类，根据模型预测对应的皮肤疾病，其识别的准确度才能保证协助医生更好的诊断病情，提高疾病的查出率[1]。

目前，在国内外的人脸识别的应用使用比较广泛的方面，主要有图像检索，街道摄像头嫌疑犯的人脸查找、出入境人脸身份的验证和人脸支付等。

## 1.2 研究意义

研究人工智能和深度学习为高级的视觉研究提供开源并且优化过的基础代码，不再需要重复造轮子。提供给开发者在此基础上进行开发的通用接口为手段传播视觉相关知识，这样代码有更强的可读性和移植性。以创造可移植的、优化过的免费开源代码来推动基于高级视觉的商业应用，这些代码可以自由使用，不要求商业应用程序开放或免费。

人工智能和深度学习的目标是为计算机视觉需要解决的问题提供工具。在某些情况下，函数库中的高级功能可以有效解决计算机视觉中的问题。即使遇到不能够一次性解决的问题，函数库中的基础组件也具有足够的完备性来增强解决方案的性能，以应对任意的计算机视觉难题[2]。

# 2 数据集

CIFAR10数据集包括60000张32 x 32的彩色图像，其中训练集50000张，测试集10000张。CIFAR10一共标注为10类，每一类图片6000张。这10类分别是airplane(飞机)，automobile（汽车），bird（鸟），cat（猫），deer（鹿），dog（狗），frog（青蛙），horse（马），ship（船）和truck（卡车），其中没有任何的重叠情况，即airplane只包括飞机，automobile只包括小型汽车，也不会在同一张照片中出现两类事物。

数据集分为五个训练批次和一个测试批次，每个批次有10000个图像。测试批次包含来自每个类别 的恰好1000个随机选择的图像。这些批次含由两个部分，第一部分是特征部分，使用一个[10000,3072的uint8的矩阵进行存储，每一行向量都是3X3大小的 3通道图片，构成的格式类似于[3,3,3]；第二部分为标签部分，使用一个10000数据的list进行存 储，每个list对应的是0-9中的一个数字，对应于物品分类。

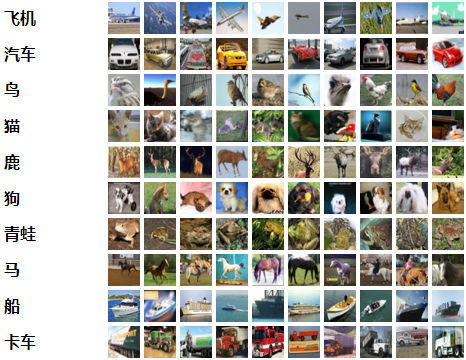
 下图是数据集中的类，以及来自每个类的10个随机图像。

图 2.1 cifar10每个分类中随机图片

# 3 研究模型

## 3.1 CNN的组成

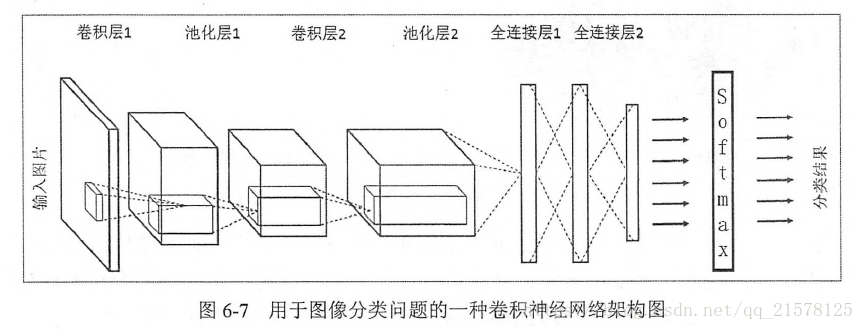
 卷积神经网络CNN(Convolutional Neural Network)，是一类深度神经网络，最常用于分析视觉图像。一个卷积神经网络通常包括输入输出层和多个隐藏层，隐藏层通常包括卷积层和RELU层（即激活函数）、池化层、全连接层和归一化层等。

图 3.1 卷积神经网络结构模型

1.输入层：CNN的输入一般是二维向量，可以有高度，比如，RGB图像。

2.卷积层：卷积层是CNN的核心，层的参数由一组可学习的滤波器（filter）或内核（kernels）组成，它们具有小的感受野，延伸到输入容积的整个深度。卷积层的作用是对输入层进行卷积，提取更高层次的特征[3]。

3.池化层：池化层（又称为下采样），它的作用是减小数据处理量同时保留有用信息，池化层的作用可以描述为模糊图像，丢掉了一些不是那么重要的特征。池化层一般包括均值池化、最大池化、高斯池化、可训练池化等[4]。

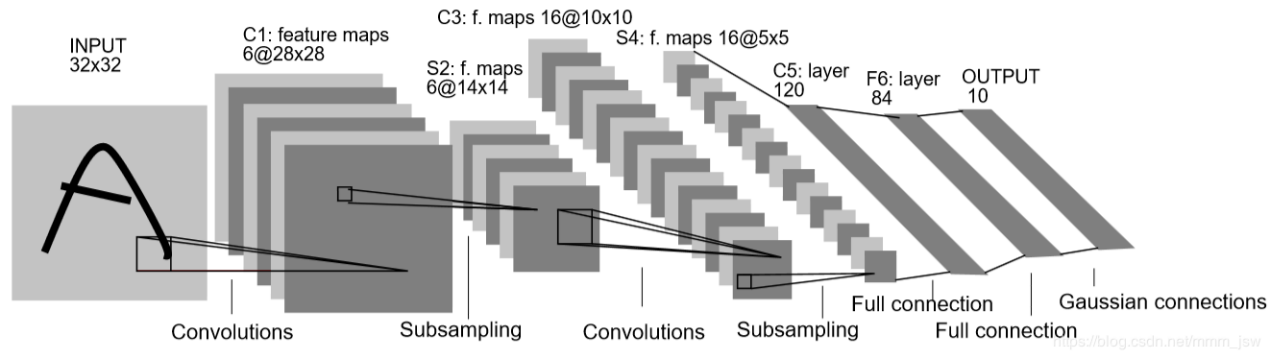
4.激活层：激活层主要是把卷积层输出结果做非线性映射，常用的激励函数有ReLU、sigmoid、tanh、LeakyReLU等。CNN采用的激励函数一般为ReLU(The Rectified Linear Unit，修正线性单元)，它的特点是收敛快，求梯度简单，但较脆弱。

5.全连接层：全连接层是一个常规的神经网络，它的作用是对经过多次卷积层和多次池化层所得出来的高级特征进行全连接（全连接就是常规神经网络的性质），算出最后的预测值。

6.输出层：输出层输出对结果的预测值，一般会加一个softmax层。

## 3.2 LENET5

LENET5源自Yann LeCun的论文“Gradient-Based Learning Applied to Document Recognition”,是一种用于手写体字符识别的非常高效的卷积神经网络。

图3.2 LENET5结构模型

LENET5网络虽然很小，但是包含了深度学习的基本模块：卷积层、池化层、全连接层。LENET5共有七层，不包含输入，每层都包含可训练参数，每个层有多个Feature Map，每个Feature Map通过一种卷积滤波器提取输入的一种特征，然后每Feature Map有多个神经元[5]。

输入：32x32的灰度图像，也就是一个通道，那么一个图像就是一个2维的矩阵，没有RGB三个通道。

1. INPUT（输入层）：32∗3232\*3232∗32的图片，共有1024个神经元。

2. C1(卷积层)：选取6个5∗55\*55∗5卷积核(不包含偏置),得到6个特征图，每个特征图的大小为32−5+1=2832-5+1=2832−5+1=28，也就是神经元的个数由1024减小到了28∗28=78428\*28=78428∗28=784 。

输入层与C1层之间的参数：6∗(5∗5+1)6\*(5\*5+1)6∗(5∗5+1) ，对于卷积层C1，每个像素都与前一层的5∗55\*55∗5个像素和1一个bias有连接，有6∗(5∗5+1)∗(28∗28)6\*(5\*5+1)\*(28\*28)6∗(5∗5+1)∗(28∗28)个连接。

3. S2（池化层）：池化层是一个下采样层，有6个14∗1414\*1414∗14的特征图，特征图中的每个单元与C1中相对应特征图的2∗22\*22∗2邻域连接。S2层每个单元对应C1中4个求和，乘以一个可训练参数，再加上一个可训练偏置。

C1与S2之间的参数： 每一个2∗22\*22∗2 求和，然后乘以一个参数，加上一个偏置，共计2∗6=122\*6=122∗6=12个参数。S2中的每个像素都与C1中的2∗22\*22∗2个像素和1个偏置相连接，所以有6∗5∗14∗14=58806\*5\*14\*14=58806∗5∗14∗14=5880个链接。

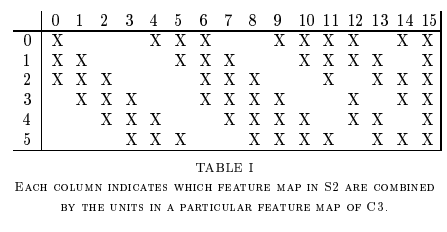
4. C3（卷积层）：选取卷积核大小为5∗55\*55∗5，得到新的图片大小为10∗1010\*1010∗10。S2与C3之间的组合：如图所示前6个feature map与S2层相连的3个feature map相连接，后面6个feature map与S2层相连的4个feature map相连接，后面3个feature map与S2层部分不相连的4个feature map相连接，最后一个与S2层的所有feature map相连。卷积核的大小依然为5∗55\*55∗5，总共有6∗(3∗5∗5+1)+6∗(4∗5∗5+1)+3∗(4∗5∗5+1)+1∗(6∗5∗5+1)=15166\*(3\*5\*5+1)+6\*(4\*5\*5+1)+3\*(4\*5\*5+1)+1\*(6\*5\*5+1)=15166∗(3∗5∗5+1)+6∗(4∗5∗5+1)+3∗(4∗5∗5+1)+1∗(6∗5∗5+1)=1516个参数。而图像大小为10∗1010\*1010∗10，所以共有151600个连接。

图 3.3连接个数

5. S4（池化层）：窗口大小为2∗22\*22∗2，有16个特征图，共有32个参数。

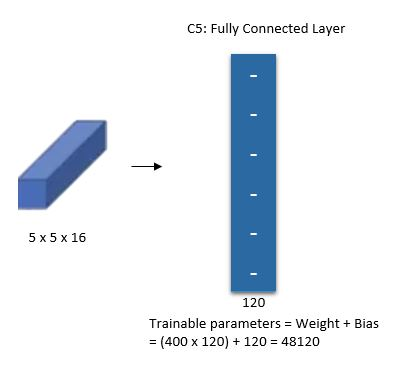
C3与S4之间的参数：16∗(25∗4+25)=200016\*(25\*4+25)=200016∗(25∗4+25)=2000个连接。6. C5（卷积层）：总共120个feature map，每个feature map与S4层所有的feature map相连接，卷积核大小为5∗55\*55∗5，而S4层的feature map的大小也是5∗55\*55∗5，所以C5的feature map就变成了一个点共计有120∗(25∗16+1)=48120120\*(25\*16+1)=48120120∗(25∗16+1)=48120个参数。

图 3.4 参数计算

7. F6（全连接层）：F6相当于MLP（Multi-Layer Perceptron，多层感知机）中的隐含层，有84个节点，所以有84∗(120+1)=1016484\*(120+1)=1016484∗(120+1)=10164个参数，F6采用了sigmoid函数。

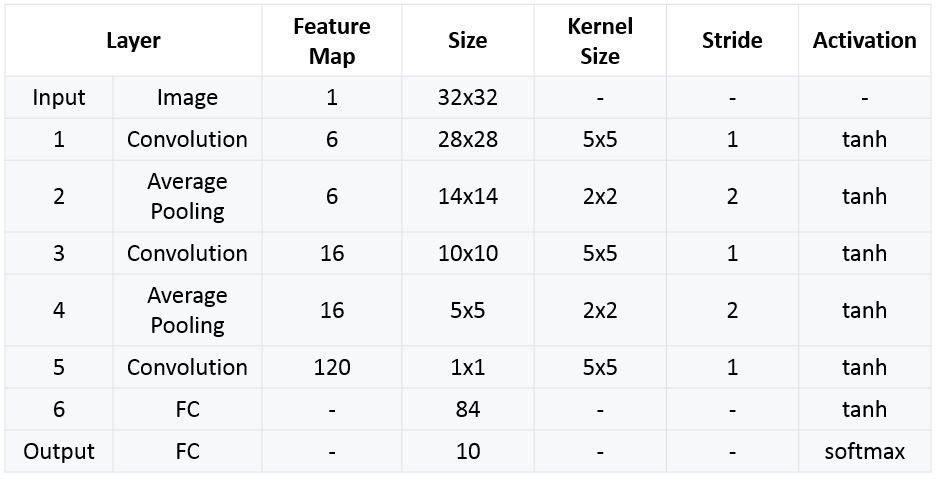
8. Output（输出层）：全连接层，共有10个节点，采用的是径向基函数（RBF）的网络连接方式。

图 3.5 LENET5总结

## 3.3 VGG16模型

卷积神经网络的输入是一个固定大小的224x224的RGB图像，唯一做的预处理是将图像每一个像素值减去RGB三通道的均值。输入图像经过一系列堆叠的卷积层，使用小感受野的3x3卷积核，卷积的步长固定为1。在部分网络结构设计中，包含大小为1x1的卷积核，相当于对输入做非线性变换。网络中包含5个最大池化层，池化作用在2x2的区域上，步长为2。网络的最后包含三个全连接层，前两个全连接层包含4096个神经元，最后一层是1000个神经元对应ILSVRC分类任务的1000个类别，最后是softmax层。所有的隐藏层都使用的是Relu非线性激活函数[6]。

### 3.3.1 VGG

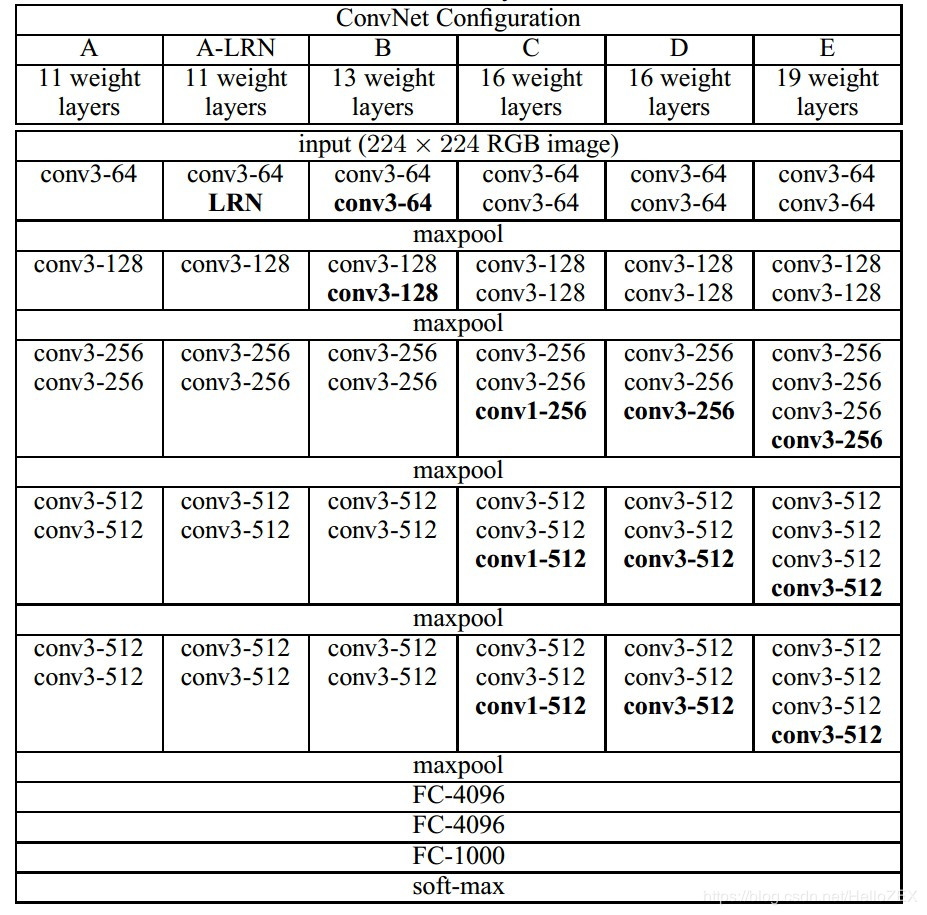
 VGG卷积神经网络是牛津大学在2014年提出来的模型。当这个模型被提出时，由于它的简洁性和实用性，马上成为了当时最流行的卷积神经网络模型。它在图像分类和目标检测任务中都表现出非常好的结果。

图3.6 VGG结构图

上图中，每一列对应一种结构配置。例如，图中D部分即指明了VGG16所采用的结构。我们针对VGG16进行具体分析发现，VGG16共包含：13个卷积层（Convolutional Layer），分别用conv3-XXX表示、3个全连接层（Fully connected Layer）,分别用FC-XXXX表示、5个池化层（Pool layer）,分别用maxpool表示。其中，卷积层和全连接层具有权重系数，因此也被称为权重层，总数目为13+3=16，这即是VGG16中16的来源。(池化层不涉及权重，因此不属于权重层，不被计数)。

### 3.3.2 VGG16结构

图3.7 VGG16结构图

VGG网络的具体配置：

1、输入图像为224x224x3；

2、经过第一个卷积层filter: 3x3x64,padding=1,stride=1,得到224x224x64的feature map，参数数量为3x3x3x64+64=1792，左右各padding1个像素来确保卷积后特征图大小不变；

3、再经过第二个卷积层filter: 3x3x64,padding=1,stride=1,得到224x224x64的feature map，参数数量为3x3x64x64+64=36928;

4、再经过最大池化层：2x2,stride=2,padding=0,得到112x112x64的feature map，不产生参数；

5、再经过第三个卷积层filter: 3x3x128,padding=1,stride=1,得到112x112x128的feature map，参数数量3x3x64x128+128=73856；

6、经过FC1层：输入为7x7x512,输出4096，就是讲7x7x512的feature map压缩为4096x1的矩阵，参数数量为:7x7x512x4096+1；有两个全连接层，每层4096个神经元（想起来面试的一个问题，为什么CNN后接的全连接层大多是2层，而不是1层，3层？理论上2层的神经网络可以拟合任意函数，1层只能拟合线性函数，多加一层3层参数量会大大提升，会增强4096\*\*2+4096个参数，综合考虑模型性能和参数量，所以一般设置为2层全连接）；

7、再经过FC2和FC3,最终得到1000x1的矩阵输出，对应softmax的1000个标签。激活函数全部使用ReLU；

VGG16是一种只需要专注于构建卷积层的简单网络。首先用3×3，步幅为1的过滤器构建卷积层，padding参数为same卷积中的参数。然后用一个2×2，步幅为2的过滤器构建最大池化层。因此VGG网络的一大优点是它确实简化了神经网络结构。

## 3.4 RESNET

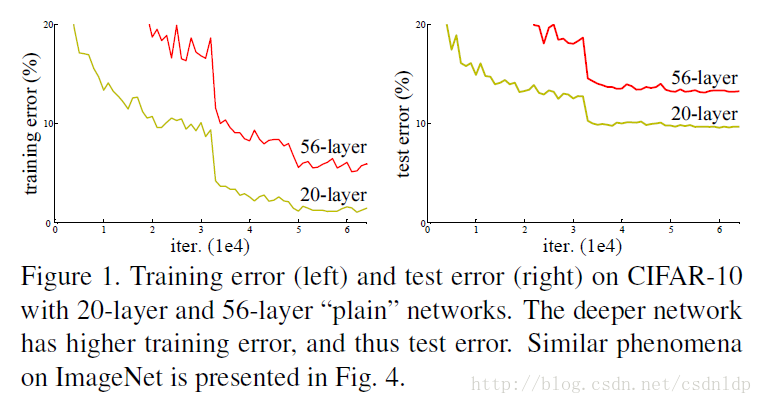
 深度卷积网络在图像分类任务上取得了一系列突破。深度网络通过多层端到端的方式，集成了低中高三个层次的特征和分类器，并且这些特征的数量还可以通过堆叠层数来增加。在ImageNet数据集上获胜的网络揭示了网络深度的重要性。随着网络层数的增加，训练的问题随之凸显。比较显著的问题有梯度消失/爆炸，这会在一开始就影响收敛。收敛的问题可以通过正则化来得到部分的解决。在深层网络能够收敛的前提下，随着网络深度的增加，正确率开始饱和甚至下降，称之为网络的退化(degradation)问题[7]。示例可见下图，显然56层的网络相对于20层的网络，不管是训练误差还是测试误差都显著增大。

图 3.8 RESNET20层与56层误差对比

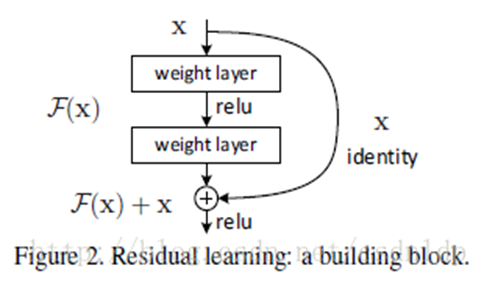
 在给定的网络上增加层数会增大训练误差。 网络的退化说明不是所有的系统都很容易优化。考虑一个浅层的网络架构和在它基础上构建的深层网络，在极端条件下，如果增加的所有层都是前一层的直接复制（即y=x），这种情况下深层网络的训练误差应该和浅层网络相等。因此，网络退化的根本原因还是优化问题。 为了解决优化的难题，提出了残差网络。在残差网络中，不是让网络直接拟合原先的映射，而是拟合残差映射。 假设原始的映射为H(x)，残差网络拟合的映射为：F(x)：=H(x)[8]。

图 3.9 映射函数

下图为VGGNet-19，以及一个34层深的普通卷积神经网络，和34层深的RESNET网络的对比图。可以看到普通直连的卷积神经网络和RESNET的最大区别在于，RESNET有很多旁路的支线将输入直接连到后面的层，使得后面的层直接学习残差，这种结构也被称为shortcut或skip connections。

传统的卷积层或全连接层在信息传递时，或多或少会存在信息丢失、损耗等问题。RESNET在某种程度上解决了这个问题，通过直接将输入信息绕道传到输出，保护信息的完整性，整个网络则只需要学习输入、输出差别的那一部分，简化学习目标和难度[9]。

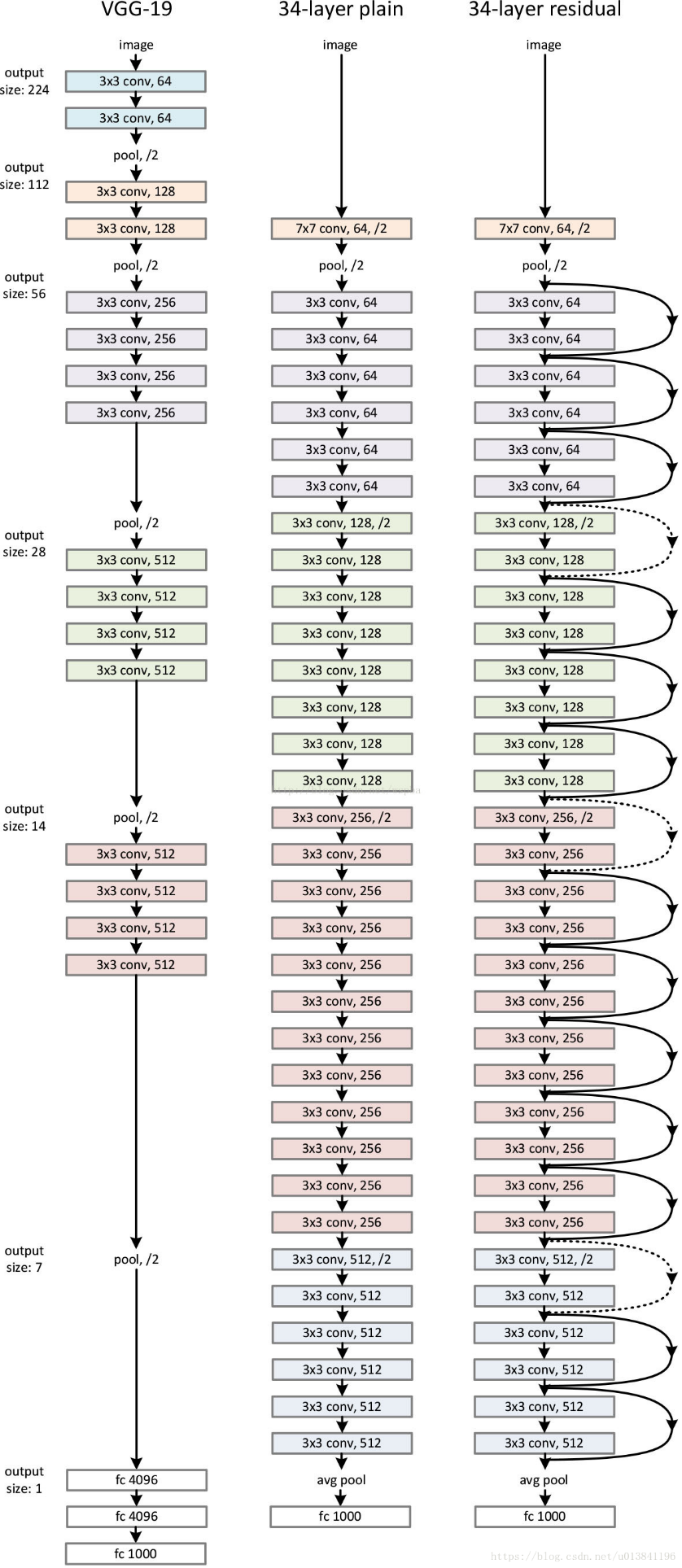


图 3.10 VGGNet-19、普通CNN、RESNET比较

# 4实验结果

在本此实验中我们使用交叉熵损失函数LOSS和训练准确率ACC来评价模型的优劣。

神经网络解决多分类问题最常用的方法是设置n个输出节点，其中n为类别的个数。对于每一个样例，神经网络可以得到的一个n维数组作为输出结果。数组中的每一个维度（也就是每一个输出节点）对应一个类别。在理想情况下，如果一个样本属于类别k，那么这个类别所对应的输出节点的输出值应该为1，而其他节点的输出都为0[10]。

最后一个输出层的节点个数与分类任务的目标数相等。假设最后的节点数为N，那么对于每一个样例，神经网络可以得到一个N维的数组作为输出结果，数组中每一个维度会对应一个类别。在最理想的情况下，如果一个样本属于k，那么这个类别所对应的的输出节点的输出值应该为1，而其他节点的输出都为0，即[0,0,1,0,….0,0]，这个数组也就是样本的Label，是神经网络最期望的输出结果，交叉熵就是用来判定实际的输出与期望的输出的接近程度[11]。

## 4.1 实验环境

本次实验过程使用MistGPU服务器作为实验环境，选用GPU为1080Ti [11G] ，RAM大小是 64G。PyTorch 版本为1.2.0，Tensorflow版本为1.13.1。

## 4.2 实验过程

### 4.2.1 LENET5实现CIFAR10分类

LENET5实现CIFAR10分类的模型建立于训练。

class Net(nn.Module): # 定义网络

def \_\_init\_\_(self, class\_num=CLASS\_NUM):

super(Net, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 6, 5)

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, class\_num)

def forward(self, x):

x = F.max\_pool2d(F.relu(self.conv1(x)), kernel\_size=(2, 2))

x = F.max\_pool2d(F.relu(self.conv2(x)), kernel\_size=(2, 2))

x = x.view(x.size()[0], -1)

x = F.relu(self.fc1(x))

x = F.relu(self.fc2(x))

x = self.fc3(x)

return x

def train(): # 训练

net = Net()

train\_dataloader, test\_dataloader, classes = getData() # 加载数据

ceterion = nn.CrossEntropyLoss() # 交叉熵损失

optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

for epoch in range(MAX\_EPOCH):

for step, data in enumerate(train\_dataloader):

inputs, labels = data

inputs, labels = Variable(inputs), Variable(labels)

optimizer.zero\_grad()

outputs = net(inputs)

loss = ceterion(outputs, labels)

loss.backward()

optimizer.step()

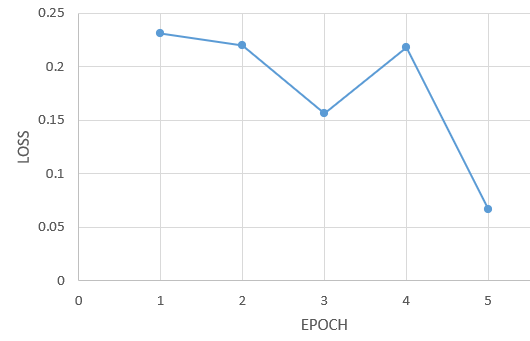
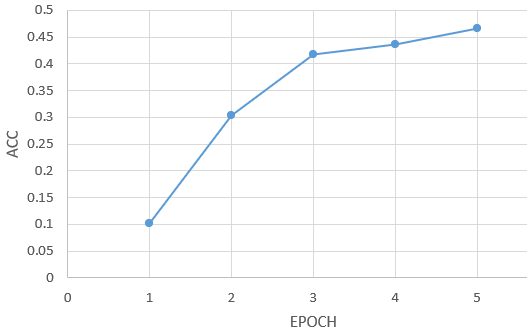
if step % 3000 == 39:

acc = test\_net(net, test\_dataloader)

print('Epoch: ', epoch, ' |step: ', step, ' |train\_loss: ', loss.item(),

'|test accuracy:%.4f' % acc)

实验结果如下，其中epoch=5, 生成每个 Epoch与LOSS和ACC的函数图像。

 图4.1 LENET5实现CIFAR10实验结果

在LENET5模型下CIFAR10数据在5次训练中以交叉熵损失函数LOSS和训练准确率ACC为评价标准下LOSS接近于0.1，其训练精度达到了46%。相对偏低。

### 4.2.2 VGG16实现CIFAR10分类

VGG16实现CIFAR10分类模型构建。

class Net(nn.Module):

def \_\_init\_\_(self):

super(Net,self).\_\_init\_\_()

# 输入shape 3\*32\*32

self.conv1 = nn.Conv2d(3,64,3,padding=1) # 64\*32\*32

self.conv2 = nn.Conv2d(64,64,3,padding=1) # 64\*32\*32

self.pool1 = nn.MaxPool2d(2, 2) # 64\*16\*16

self.bn1 = nn.BatchNorm2d(64) # 64\*16\*16

self.relu1 = nn.ReLU() # 64\*16\*16

self.conv3 = nn.Conv2d(64,128,3,padding=1) # 128\*16\*16

self.conv4 = nn.Conv2d(128, 128, 3,padding=1) # 128\*16\*16

self.pool2 = nn.MaxPool2d(2, 2, padding=1) # 128\*9\*9

self.bn2 = nn.BatchNorm2d(128) # 128\*9\*9

self.relu2 = nn.ReLU() # 128\*9\*9

self.conv5 = nn.Conv2d(128,128, 3,padding=1) # 128\*9\*9

self.conv6 = nn.Conv2d(128, 128, 3,padding=1) # 128\*9\*9

self.conv7 = nn.Conv2d(128, 128, 1,padding=1) # 128\*11\*11

self.pool3 = nn.MaxPool2d(2, 2, padding=1) # 128\*6\*6

self.bn3 = nn.BatchNorm2d(128) # 128\*6\*6

self.relu3 = nn.ReLU() # 128\*6\*6

self.conv8 = nn.Conv2d(128, 256, 3,padding=1) # 256\*6\*6

self.conv9 = nn.Conv2d(256, 256, 3, padding=1) # 256\*6\*6

self.conv10 = nn.Conv2d(256, 256, 1, padding=1) # 256\*8\*8

self.pool4 = nn.MaxPool2d(2, 2, padding=1) # 256\*5\*5

self.bn4 = nn.BatchNorm2d(256) # 256\*5\*5

self.relu4 = nn.ReLU() # 256\*5\*5

self.conv11 = nn.Conv2d(256, 512, 3, padding=1) # 512\*5\*5

self.conv12 = nn.Conv2d(512, 512, 3, padding=1) # 512\*5\*5

self.conv13 = nn.Conv2d(512, 512, 1, padding=1) # 512\*7\*7

self.pool5 = nn.MaxPool2d(2, 2, padding=1) # 512\*4\*4

self.bn5 = nn.BatchNorm2d(512) # 512\*4\*4

self.relu5 = nn.ReLU() # 512\*4\*4

self.fc14 = nn.Linear(512\*4\*4,1024) # 1\*1024

self.drop1 = nn.Dropout2d() # 1\*1024

self.fc15 = nn.Linear(1024,1024) # 1\*1024

self.drop2 = nn.Dropout2d() # 1\*1024

self.fc16 = nn.Linear(1024,10) # 1\*10

def forward(self,x):

x = x.to(device) # 自加

x = self.conv1(x)

x = self.conv2(x)

x = self.pool1(x)

x = self.bn1(x)

x = self.relu1(x)

x = self.conv3(x)

x = self.conv4(x)

x = self.pool2(x)

x = self.bn2(x)

x = self.relu2(x)

x = self.conv5(x)

x = self.conv6(x)

x = self.conv7(x)

x = self.pool3(x)

x = self.bn3(x)

x = self.relu3(x)

x = self.conv8(x)

x = self.conv9(x)

x = self.conv10(x)

x = self.pool4(x)

x = self.bn4(x)

x = self.relu4(x)

x = self.conv11(x)

x = self.conv12(x)

x = self.conv13(x)

x = self.pool5(x)

x = self.bn5(x)

x = self.relu5(x)

# print(" x shape ",x.size())

x = x.view(-1,512\*4\*4)

x = F.relu(self.fc14(x))

x = self.drop1(x)

x = F.relu(self.fc15(x))

x = self.drop2(x)

x = self.fc16(x)

return x

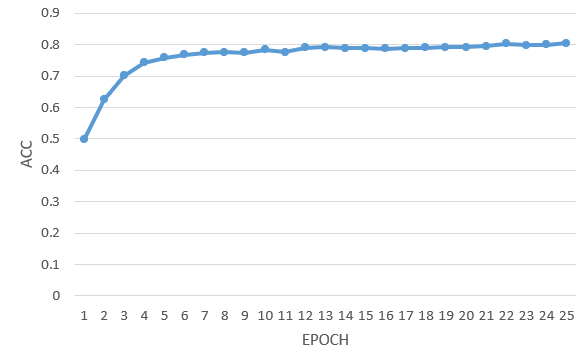
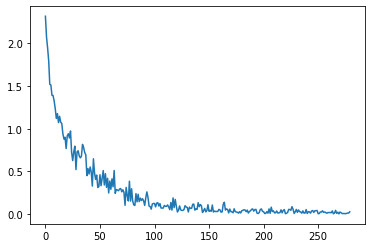
实验结果如下，其中epoch=25, 生成每个 Epoch与LOSS和ACC的函数图像。

图4.2 VGG16实现CIFAR10实验结果

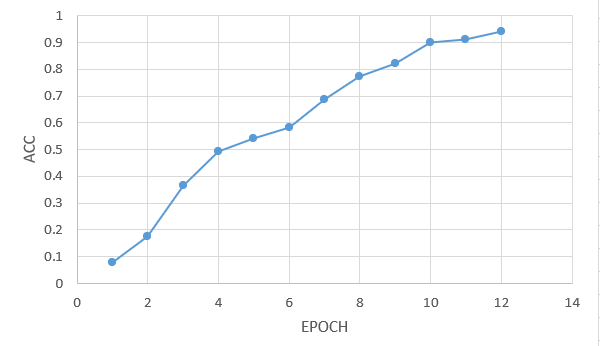
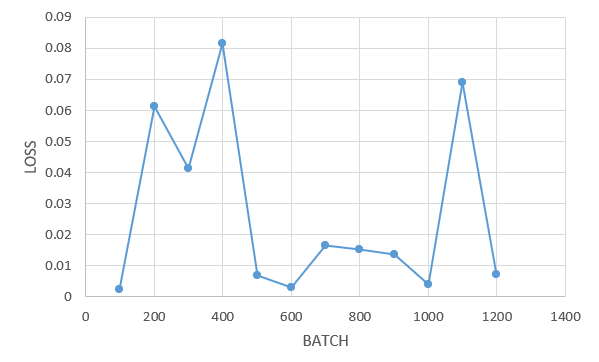
在VGG16模型下CIFAR10数据在25次训练中以交叉熵损失函数LOSS和训练准确率ACC为评价标准下LOSS接近于0.1，其训练精度达到了80.3%。

### 4.2.3 RESNET实现CIFAR10分类

RESNET实现CIFAR10分类训练模型。

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

实验结果如下，其中epoch=12, 生成每个 Epoch与LOSS和ACC的函数图像

图4.3 RESNAT实现CIFAR10实验结果

RESNET网络初始的时候训练精度较低，随着网络层次的加深，训练精度也升高，损失函数随之降低。训练准确率ACC为评价标准下LOSS接近于0.1，其训练精度达到了94.17%。

## 4.3 结果对比

在三种模型LENET5、VGG16、RESNET下， CIFAR10的交叉熵损失函数LOSS数值均接近0.1，但是准确率却是不一样，因此在本次结果比较中我们对精确度做比较。下表是两个数据集在两个模型下的精确度汇总。

表4.1 准确率

|  |  |  |  |
| --- | --- | --- | --- |
|  | LENET5 | VGG16 | RESNET |
| CIFAR10 | 46.58% | 80.3% | 94.17 |

在基于LENET5模型、VGG16模型、RESNET模型下对数据集的训练和测试过程中， VGG16对CIFAR10的准确率可以达到80.3%，RESNET对CIFAR10的准确率最高达到94.17%，LENET5对CIFAR10的交叉熵损失函数值虽然较低，但是分类识别结果差强人意。

LENET5网络结构比较完整，包括卷积层、pooling层、全连接层，网络结构小而全，是一种用于手写体字符识别的非常高效的卷积神经网络。VGG16卷积层、池化层均采用相同的卷积核参数，模型是由若干卷积层和池化层堆叠（stack）的方式构成，比较容易形成较深的网络结构。

由于CIFAR10数据集更为复杂，因此使用简单的网络模型预测的准确率并不高。改进网络的能力，主要是选择更深层的网络结构，选用RESNET网络结构，经过测试，其表现结果较好。

# 5总 结

学习人工智能和深度学习首先将所要解决的问题抽象成机器学习的问题，目标问题到底是一个什么问题，分类，回归还是聚类问题，要达到的目标是什么，对准确率的要求等。其次、明确该项目可以获取哪些数据，大概有多少样本，多少个特征，训练的时候内存占用量，需不需要用分布式，接着、考虑对特征数据做处理和选择，包括特征筛选，数据清晰，归一化，缺失值处理......,然后利用特征选择的方法还可以进一步用PCA,LDA等方法降维。然后就是模型选择、训练与调优，根据样本大小特征选择相应模型，然后对参数进行调优，随后还需要模型诊断，进行特征融合，模型融合等进一步提高预测效果。

在基于LENET5模型、VGG16模型、RESNET模型下对数据集的训练和测试过程中，选用更深层次的网络其训练和测试精度越高，当然也可以改变激活函数、损失函数等改进网络模型。

# 参考文献

[1]李杰骏.基于卷积神经网络的多样式简体字识别[J].电脑迷,2018(12):227.

[2]Bruno Roberto,CENTENO PEREZ,姜成友.基于深度学习的图像识别研究[J].现代经济信息,2018(23):419.

[3]张涛,杨剑,宋文爱,郭雁蓉.改进卷积神经网络模型设计方法[J].计算机工程与设计,2019,40(07):1885-1890.

[4]邓璐芗,许鑫.数字人文人工智能平台的设计与实现——以ECNU-DHAI平台为例[J/OL].图书馆杂志:1-10[2020-06-30].http://kns.cnki.net/kcms/detail/31.1108.G2.20200628.1836.006.html.

[5]王培进,杜晓林.基于深度学习的复杂系统仿人思维控制研究[J/OL].烟台大学学报(自然科学与工程版):1-10[2020-06-30].https://doi.org/10.13951/j.cnki.37-1213/n.190309.

[6]王菽裕,吴思,靳玮琨,关琛夕,王蒙,毕冰洁.基于VGG16架构的中国名人面孔识别[J].科技风,2020(07):24.

[7]倪照风,马原东,崔潇,郦烜杰,杨秀璋,罗子江.ResNet网络下教室人物姿态分类[J].现代电子技术,2020,43(12):42-46.

[8]马素刚,赵祥模,侯志强,王忠民,孙韩林.一种基于ResNet网络特征的视觉目标跟踪算法[J/OL].北京邮电大学学报:1-7[2020-06-30].https://doi.org/10.13190/j.jbupt.2019-071.

[9]袁嘉杰,张灵,陈云华.基于注意力卷积模块的深度神经网络图像识别[J].计算机工程与应用,2019,55(08):9-16.

[10]Emma Lejeune. Mechanical MNIST: A benchmark dataset for mechanical metamodels[J]. Elsevier Ltd,2020,36.

[11]. Engineering; Reports on Engineering from Chongqing University of Posts and Telecommunications Provide New Insights (Crack Detection of Concrete Pavement With Cross-entropy Loss Function and Improved Vgg16 Network Model)[J]. Mathematics Week,2020.

# 附录

1 LENET5实现

import torchvision as tv

import torch.nn as nn

import torch as t

from torch.autograd import Variable

import torch.nn.functional as F

import torchvision.transforms as transforms

from torch.utils.data import DataLoader

from torch import optim

MAX\_EPOCH = 2

CLASS\_NUM = 10

class Net(nn.Module): # 定义网络

def \_\_init\_\_(self, class\_num=CLASS\_NUM):

super(Net, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 6, 5)

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, class\_num)

def forward(self, x):

x = F.max\_pool2d(F.relu(self.conv1(x)), kernel\_size=(2, 2))

x = F.max\_pool2d(F.relu(self.conv2(x)), kernel\_size=(2, 2))

x = x.view(x.size()[0], -1)

x = F.relu(self.fc1(x))

x = F.relu(self.fc2(x))

x = self.fc3(x)

return x

def getData(): # 数据预处理

transform = transforms.Compose([transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# 训练集

train\_set = tv.datasets.CIFAR10(root='/data/', train=True, transform=transform, download=True)

train\_loader = DataLoader(train\_set, batch\_size=4, shuffle=True)

# 测试集

test\_set = tv.datasets.CIFAR10(root='/data/', train=False, transform=transform, download=True)

test\_loader = t.utils.data.DataLoader(test\_set, batch\_size=4, shuffle=False)

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

return train\_loader, test\_loader, classes

def train(): # 训练

net = Net()

train\_dataloader, test\_dataloader, classes = getData() # 加载数据

ceterion = nn.CrossEntropyLoss() # 交叉熵损失

optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

for epoch in range(MAX\_EPOCH):

for step, data in enumerate(train\_dataloader):

inputs, labels = data

inputs, labels = Variable(inputs), Variable(labels)

optimizer.zero\_grad()

outputs = net(inputs)

loss = ceterion(outputs, labels)

loss.backward()

optimizer.step()

if step % 3000 == 39:

acc = test\_net(net, test\_dataloader)

print('Epoch: ', epoch, ' |step: ', step, ' |train\_loss: ', loss.item(),

'|test accuracy:%.4f' % acc)

print('Finished Training')

return net

def test\_net(net, test\_dataloader): #  获取在测试集上的准确率

correct, total = .0, .0

for inputs, label in test\_dataloader:

output = net(inputs)

\_, predicted = t.max(output, 1) #  获取分类结果

total += label.size(0) # 记录总个数

correct += (predicted == label).sum() #  记录分类正确的个数

return float(correct) / total

if \_\_name\_\_ == '\_\_main\_\_':

net = train()

2 GVGG16实现

import torch

import torchvision

from torch.autograd import Variable

import torch.nn as nn

import torch.nn.functional as F

import torch.optim as optim

import torch.utils.data as Data

import torchvision.transforms as transforms

import numpy as np

from PIL import Image

import matplotlib.pyplot as plt

import time

import os

#from torchsummary import summary

# 预设参数

CLASS\_NUM = 10

BATCH\_SIZE = 128

EPOCH =25

# 检验GPU是否可用

device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu")

print(device)

class Net(nn.Module):

def \_\_init\_\_(self):

super(Net,self).\_\_init\_\_()

# 输入shape 3\*32\*32

self.conv1 = nn.Conv2d(3,64,3,padding=1) # 64\*32\*32

self.conv2 = nn.Conv2d(64,64,3,padding=1) # 64\*32\*32

self.pool1 = nn.MaxPool2d(2, 2) # 64\*16\*16

self.bn1 = nn.BatchNorm2d(64) # 64\*16\*16

self.relu1 = nn.ReLU() # 64\*16\*16

self.conv3 = nn.Conv2d(64,128,3,padding=1) # 128\*16\*16

self.conv4 = nn.Conv2d(128, 128, 3,padding=1) # 128\*16\*16

self.pool2 = nn.MaxPool2d(2, 2, padding=1) # 128\*9\*9

self.bn2 = nn.BatchNorm2d(128) # 128\*9\*9

self.relu2 = nn.ReLU() # 128\*9\*9

self.conv5 = nn.Conv2d(128,128, 3,padding=1) # 128\*9\*9

self.conv6 = nn.Conv2d(128, 128, 3,padding=1) # 128\*9\*9

self.conv7 = nn.Conv2d(128, 128, 1,padding=1) # 128\*11\*11

self.pool3 = nn.MaxPool2d(2, 2, padding=1) # 128\*6\*6

self.bn3 = nn.BatchNorm2d(128) # 128\*6\*6

self.relu3 = nn.ReLU() # 128\*6\*6

self.conv8 = nn.Conv2d(128, 256, 3,padding=1) # 256\*6\*6

self.conv9 = nn.Conv2d(256, 256, 3, padding=1) # 256\*6\*6

self.conv10 = nn.Conv2d(256, 256, 1, padding=1) # 256\*8\*8

self.pool4 = nn.MaxPool2d(2, 2, padding=1) # 256\*5\*5

self.bn4 = nn.BatchNorm2d(256) # 256\*5\*5

self.relu4 = nn.ReLU() # 256\*5\*5

self.conv11 = nn.Conv2d(256, 512, 3, padding=1) # 512\*5\*5

self.conv12 = nn.Conv2d(512, 512, 3, padding=1) # 512\*5\*5

self.conv13 = nn.Conv2d(512, 512, 1, padding=1) # 512\*7\*7

self.pool5 = nn.MaxPool2d(2, 2, padding=1) # 512\*4\*4

self.bn5 = nn.BatchNorm2d(512) # 512\*4\*4

self.relu5 = nn.ReLU() # 512\*4\*4

self.fc14 = nn.Linear(512\*4\*4,1024) # 1\*1024

self.drop1 = nn.Dropout2d() # 1\*1024

self.fc15 = nn.Linear(1024,1024) # 1\*1024

self.drop2 = nn.Dropout2d() # 1\*1024

self.fc16 = nn.Linear(1024,10) # 1\*10

def forward(self,x):

x = x.to(device) # 自加

x = self.conv1(x)

x = self.conv2(x)

x = self.pool1(x)

x = self.bn1(x)

x = self.relu1(x)

x = self.conv3(x)

x = self.conv4(x)

x = self.pool2(x)

x = self.bn2(x)

x = self.relu2(x)

x = self.conv5(x)

x = self.conv6(x)

x = self.conv7(x)

x = self.pool3(x)

x = self.bn3(x)

x = self.relu3(x)

x = self.conv8(x)

x = self.conv9(x)

x = self.conv10(x)

x = self.pool4(x)

x = self.bn4(x)

x = self.relu4(x)

x = self.conv11(x)

x = self.conv12(x)

x = self.conv13(x)

x = self.pool5(x)

x = self.bn5(x)

x = self.relu5(x)

# print(" x shape ",x.size())

x = x.view(-1,512\*4\*4)

x = F.relu(self.fc14(x))

x = self.drop1(x)

x = F.relu(self.fc15(x))

x = self.drop2(x)

x = self.fc16(x)

return x

def unpickle(file):

import pickle

with open(file, 'rb') as fo:

dict = pickle.load(fo, encoding='bytes')

return dict

# 从源文件读取数据

# 返回 train\_data[50000,3072]和labels[50000]

# test\_data[10000,3072]和labels[10000]

def get\_data(train=False):

data = None

labels = None

if train == True:

for i in range(1, 6):

batch = unpickle('data/cifar-10-batches-py/data\_batch\_' + str(i))

if i == 1:

data = batch[b'data']

else:

data = np.concatenate([data, batch[b'data']])

if i == 1:

labels = batch[b'labels']

else:

labels = np.concatenate([labels, batch[b'labels']])

else:

batch = unpickle('data/cifar-10-batches-py/test\_batch')

data = batch[b'data']

labels = batch[b'labels']

return data, labels

# 对于彩色图像，色彩通道不存在平稳特性

transform = transforms.Compose([

# ToTensor是指把PIL.Image(RGB) 或者numpy.ndarray(H \* W \* C)

# 从0到255的值映射到0到1的范围内，并转化成Tensor格式。

transforms.ToTensor(),

# Normalize函数将图像数据归一化到[-1,1]

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

]

)

# 将标签转换为torch.LongTensor

def target\_transform(label):

label = np.array(label) # 变为ndarray

target = torch.from\_numpy(label).long() # 变为torch.LongTensor

return target

# 数据集

class Cifar10\_Dataset(Data.Dataset):

def \_\_init\_\_(self, train=True, transform=None, target\_transform=None):

# 初始化文件路径

self.transform = transform

self.target\_transform = target\_transform

self.train = train

# 载入训练数据集

if self.train:

self.train\_data, self.train\_labels = get\_data(train)

self.train\_data = self.train\_data.reshape((50000, 3, 32, 32))

# 将图像数据格式转换为[height,width,channels]方便预处理

self.train\_data = self.train\_data.transpose((0, 2, 3, 1))

# 载入测试数据集

else:

self.test\_data, self.test\_labels = get\_data()

self.test\_data = self.test\_data.reshape((10000, 3, 32, 32))

self.test\_data = self.test\_data.transpose((0, 2, 3, 1))

pass

# 从数据集中读取一个数据并对数据进行预处理返回一个数据对，如（data,label）

def \_\_getitem\_\_(self, index):

if self.train:

img, label = self.train\_data[index], self.train\_labels[index]

else:

img, label = self.test\_data[index], self.test\_labels[index]

img = Image.fromarray(img)

# 图像预处理

if self.transform is not None:

img = self.transform(img)

# 标签预处理

if self.target\_transform is not None:

target = self.target\_transform(label)

return img, target

def \_\_len\_\_(self):

# 返回数据集的size

if self.train:

return len(self.train\_data)

else:

return len(self.test\_data)

if \_\_name\_\_ == '\_\_main\_\_':

# 读取训练集和测试集

train\_data = Cifar10\_Dataset(True, transform, target\_transform)

print('size of train\_data:{}'.format(train\_data.\_\_len\_\_()))

test\_data = Cifar10\_Dataset(False, transform, target\_transform)

print('size of test\_data:{}'.format(test\_data.\_\_len\_\_()))

train\_loader = Data.DataLoader(dataset=train\_data, batch\_size=BATCH\_SIZE, shuffle=True)

net = Net()

net.to(device)

# ------------------------------------------------

with torch.no\_grad():

for input\_data, \_ in train\_loader:

break

# summary(model.to(hyperparams['device']), input.size()[1:], device=hyperparams['device'])

# print(input\_data.size())

#summary(net, input\_data.size()[1:])

os.system('pause')

# ------------------------------------------------

# 定义优化器

optimizer = optim.SGD(net.parameters(), lr=0.001, 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）

# 定义损失函数

# 在使用CrossEntropyLoss时target直接使用类别索引，不适用one-hot

loss\_fn = nn.CrossEntropyLoss()

loss\_list = []

Accuracy = []

for epoch in range(1, EPOCH + 1):

# 训练部分

timestart = time.time() # 自加计时

for step, (x, y) in enumerate(train\_loader):

b\_x = Variable(x)

b\_y = Variable(y)

output = net(b\_x)

b\_x, b\_y = b\_x.to(device), b\_y.to(device) # CPU 转 GPU

loss = loss\_fn(output, b\_y)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

# 记录loss

if step % 50 == 0:

loss\_list.append(loss)

# 每完成一个epoch进行一次测试观察效果

pre\_correct = 0.0

test\_loader = Data.DataLoader(dataset=test\_data, batch\_size=100, shuffle=True)

for (x, y) in (test\_loader):

b\_x = Variable(x)

b\_y = Variable(y)

b\_x, b\_y = b\_x.to(device), b\_y.to(device) # 自加

output = net(b\_x)

pre = torch.max(output, 1)[1]

# pre\_correct = pre\_correct.to(device) # 自加

pre\_correct = pre\_correct + float(torch.sum(pre == b\_y))

print('EPOCH:{epoch},ACC:{acc}%'.format(epoch=epoch, acc=(pre\_correct / float(10000)) \* 100))

Accuracy.append(pre\_correct / float(10000) \* 100)

# 自加计时

print('epoch %d cost %3f sec' % (epoch, time.time() - timestart))

# 保存网络模型

torch.save(net, 'lenet\_cifar\_10.model')

# 绘制loss变化曲线

plt.figure()

plt.plot(loss\_list)

plt.figure()

plt.plot(Accuracy)

plt.show()

3 RESNAT实现

import torch

import torch.nn as nn

import torch.nn.functional as F

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

import argparse

import os

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()

# resnet就是这样设计的啊，你可以看第一张网络结构图，

# shortcut都是加在通道和输出通道不相等的地方，

# 如果输入输出通道相同，就都是加在步长不为1的地方

if stride != 1 or inchannel != outchannel:

# print(stride,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):

# print(type(ResidualBlock)) 用类进行初始化

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)

# ResNet18 18 =1(初始卷积）+4\*4+1（全连接）

def make\_layer(self, block, channels, num\_blocks, stride):

# print(stride)

strides = [stride] + [1] \* (num\_blocks - 1) # strides=[1,1]

# print(strides)

layers = [] # 将model children 加到列表中

for stride in strides: # 每次重复两次残差块，总共4层卷积层参数

layers.append(block(self.inchannel, channels, stride))

self.inchannel = channels

return nn.Sequential(\*layers) # 返回所有model.childen

# inchanels: 64--->64--->128--->256

# outchanels:64--->128--->256--->512

# strides[1,1]--->[2,1]--->[2,1]--->[2,1]

def forward(self, x):

out = self.conv1(x) # shape [2, 64, 32, 32]

# print(out.shape)

out = self.layer1(out) # shape 第一次 inchannel ==outchannel,stride ==1,不使用残差块

# shape [2, 64, 32, 32]

# print(out.shape)

out = self.layer2(out)

# print(out.shape) [2, 128, 16, 16]

out = self.layer3(out)

# print(out.shape)[2, 256, 8, 8]

out = self.layer4(out)

# print(out.shape)[2, 512, 4, 4]

out = F.avg\_pool2d(out, 4)

# print(out.shape)[2, 512, 1, 1]

out = out.view(out.size(0), -1)

# print(out.shape)shape[2, 512]

out = self.fc(out)

# print(out.shape)torch.Size([2, 10])

return out

def ResNet18():

return ResNet(ResidualBlock)

# 训练

if \_\_name\_\_ == "\_\_main\_\_":

# 定义是否使用GPU

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# 超参数设置

EPOCH = 135 # 遍历数据集次数

pre\_epoch = 0 # 定义已经遍历数据集的次数

BATCH\_SIZE = 128 # 批处理尺寸(batch\_size)

LR = 0.01 # 学习率

# 准备数据集并预处理

transform\_train = 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

])

transform\_test = 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=transform\_train) # 训练数据集

trainloader = 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 = torchvision.datasets.CIFAR10(root='./data/', train=False, download=True, transform=transform\_test)

testloader = torch.utils.data.DataLoader(testset, batch\_size=100, shuffle=False, num\_workers=2)

# Cifar-10的标签

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

# 模型定义-ResNet

net = ResNet18().to(device)

# 定义损失函数和优化方式

m = './model/'

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）

best\_acc = 85 # 2 初始化best test accuracy

print("Start Training, Resnet-18!") # 定义遍历数据集的次数

with open("acc.txt", "w") as f:

with open("log.txt", "w")as f2:

for epoch in range(pre\_epoch, EPOCH):

print('\nEpoch: %d' % (epoch + 1))

net.train()

sum\_loss = 0.0

correct = 0.0

total = 0.0

for i, data in enumerate(trainloader, 0):

# 准备数据

length = len(trainloader)

inputs, labels = data

inputs, labels = inputs.to(device), labels.to(device)

optimizer.zero\_grad()

# forward + backward

outputs = net(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

# 每训练1个batch打印一次loss和准确率

sum\_loss += loss.item()

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += predicted.eq(labels.data).cpu().sum()

print('[epoch:%d, iter:%d] Loss: %.03f | Acc: %.3f%% '

% (epoch + 1, (i + 1 + epoch \* length), sum\_loss / (i + 1), 100. \* correct / total))

f2.write('%03d %05d |Loss: %.03f | Acc: %.3f%% '

% (epoch + 1, (i + 1 + epoch \* length), sum\_loss / (i + 1), 100. \* correct / total))

f2.write('\n')

f2.flush()

# 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 testloader:

net.eval()

images, labels = data

images, labels = images.to(device), labels.to(device)

outputs = net(images)

# 取得分最高的那个类 (outputs.data的索引号)

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum()

print('测试分类准确率为：%.3f%%' % (100 \* correct / total))

acc = 100. \* correct / total

# 将每次测试结果实时写入acc.txt文件中

print('Saving model......')

torch.save(net.state\_dict(), '%s/net\_%03d.pth' % (m, epoch + 1))

f.write("EPOCH=%03d,Accuracy= %.3f%%" % (epoch + 1, acc))

f.write('\n')

f.flush()

# 记录最佳测试分类准确率并写入best\_acc.txt文件中

if acc > best\_acc:

f3 = open("best\_acc.txt", "w")

f3.write("EPOCH=%d,best\_acc= %.3f%%" % (epoch + 1, acc))

f3.close()

best\_acc = acc

print("Training Finished, TotalEPOCH=%d" % EPOCH)