

**模式识别与机器学习**

**实验报告**

**班级：10051901**

**姓名：牛远卓**

**学号：2019302973**

1. **实验目的**

通过跑神经网络模型，了解它的构造、特点以及思想。

1. **实验原理**

Artificial Neural Network, 缩写ANN, 简称为神经网络，在机器学习，尤其是深度学习领域广泛应用。

神经网络采用了一种仿生学的思想，通过模拟生物神经网络的结构和功能来实现建模。借鉴神经元这一生物结构，1943年MoCulloch和Pitts提出了人工神经元模型，即M-P神经元模型。

ALexnet介绍

Alexnet模型由5个卷积层和3个池化Pooling 层 ，其中还有3个全连接层构成。AlexNet 跟 LeNet 结构类似，但使⽤了更多的卷积层和更⼤的参数空间来拟合⼤规模数据集 ImageNet。它是浅层神经⽹络和深度神经⽹络的分界线。

表格

中度可信度描述已自动生成

图示

描述已自动生成

Alexnet网络结构

特点：

1、在每个卷机后面添加了Relu激活函数，解决了Sigmoid的梯度消失问题，使收敛更快。

2、使用随机丢弃技术（dropout）选择性地忽略训练中的单个神经元，避免模型的过拟合（也使用数据增强防止过拟合）

3、添加了归一化LRN（Local Response Normalization，局部响应归一化）层，使准确率更高。

4、重叠最大池化（overlapping max pooling），即池化范围 z 与步长 s 存在关系 z>s 避免平均池化（average pooling）的平均效应。

Resnet18简介

1. 残差网络：（Resnet）

残差块：

让我们聚焦于神经网络局部：如图左侧所示，假设我们的原始输入为x，而希望学出的理想映射为f(x)（作为上方激活函数的输入）。左图虚线框中的部分需要直接拟合出该映射f(x)，而右图虚线框中的部分则需要拟合出残差映射f(x)−x。 残差映射在现实中往往更容易优化。 以本节开头提到的恒等映射作为我们希望学出的理想映射f(x)，我们只需将右图虚线框内上方的加权运算（如仿射）的权重和偏置参数设成0，那么f(x)即为恒等映射。 实际中，当理想映射f(x)极接近于恒等映射时，残差映射也易于捕捉恒等映射的细微波动。右图是ResNet的基础架构–残差块（residual block）。 在残差块中，输入可通过跨层数据线路更快地向前传播

图示

描述已自动生成

ResNet沿用了VGG完整的3×3卷积层设计。 残差块里首先有2个有相同输出通道数的3×3卷积层。 每个卷积层后接一个批量规范化层和ReLU激活函数。 然后我们通过跨层数据通路，跳过这2个卷积运算，将输入直接加在最后的ReLU激活函数前。 这样的设计要求2个卷积层的输出与输入形状一样，从而使它们可以相加。 如果想改变通道数，就需要引入一个额外的1×1卷积层来将输入变换成需要的形状后再做相加运算。 残差块的实现如下

**图示

描述已自动生成**

1. **实验步骤和程序流程**

数据导入

实验先将（200，15，28，28）的训练数据和（200，5，28，28）的测试数据分别通过dataload.py转换成按照类成文件夹的png图片中。再将他们通过torchvision.datasets.ImageFolder加载进网络。

神经网络

我先用的是Alexnet神经网络。我是对着Alexnet论文中的结构改的。后来，经过各种trick还是没能达到理想的结果，我使用了Resnet18，来跑模型。

整体的程序流程如下图。先用不同tricks在ALexnet上跑，后来发现结果有提升空间，使用了resnet18跑，效果拔群。

图示

描述已自动生成

1. **实验结果**

batchsize=32

epoch=32

前20个epoch lr=0.001, momentum=0.9之后的 lr=0.0005， momentum=0.9 这里的optimizer用的是SGD。

这样的效果很差

图表, 直方图

描述已自动生成

图表, 箱线图

描述已自动生成

可以看到，正确率根本没变，而且只有0.005，分明就是在猜测。而loss基本不降，只降一点点。这说明至少不是没有反传，也就是不是代码的问题，而是learning rate太小的问题。于是，我将他调大一倍，效果显著。

图表, 直方图

描述已自动生成

图表, 折线图

描述已自动生成

虽然acc与loss变化幅度增大了，但是acc数值50%不到，还是太小。值得欣慰的是暂时没看出游过拟合的迹象，于是我调大了epoch至100。

图表, 直方图

描述已自动生成

图表

描述已自动生成

可以看到，结果好了很多，但是acc在70%左右开始浮动，逐渐出现过拟合的问题。于是，我使用了l2 norm和数据增强来降低generation gap。

前100个epoch的lr=0.01, momentum=0.9，之后的50个 lr=0.007, momentum=0.9,weight\_decay=0.0001（weight\_decay是防止过拟合用的），最后50个epoch lr=0.005, momentum=0.9,weight\_decay=0.0001。其中所有optimizer都用了SGD

数据增强用了随机裁剪与随机水平翻转（由于我看大多数据集水平翻转后不破坏图像语义特征，而竖直反转会，所以就没用竖直反转）

图表, 直方图

描述已自动生成

图表

描述已自动生成

最高的时候我的acc达到了81%。

但是我听说有同学在很短时间内就能达到1，恐怖如斯。于是，我重新想了一下整个项目。发现我的数据导入存在巨大问题。原来的数据知识一个4位的数据：训练（200，15，28，28）；测试（200，5，28，28）。然而，由于我是将mat数据集先转成png图片，再通过ImageFolder导入进网络。所以，我的数据集每张图片变成彩色的RGB图像了，也就是（3，28，28）而不是（1，28，28）。这导致了我的数据集被我人工改动了，更具体的说是改复杂了三倍。所以，这对我的神经网络的训练压力加大了。

简而言之，我烦的问题是不应该先将mat数据集改成png图片，再导入网络，而是应该直接将mat数据集导入。所以，我需要重新写一下dataset类。这个类我放在后面的代码里了。这是根据dataset类的文档写的。

之后，我使用了resnet18重新训练，什么trick都没用（数据增强，动态调整learning rate等）效果非常amazng啊。

图表

描述已自动生成

不到15个epoch,测试正确率就达到了100%！！！

由此可见，虽然说一些小技巧，比如用weight\_decay,batch\_normalization,data\_augmentation等能在ALexnet的基础上提升正确率，但是非常有限。使用深度大大增加的Resnet18,并用没改动的源数据重新训练后，效果发生了翻天覆地的变化。这说明深度网络越深，层数越多，越有利于模型学习到语义信息。

**附1:参考文献**

1. <https://baijiahao.baidu.com/s?id=1700872420698221621&wfr=spider&for=pc>
2. <https://blog.csdn.net/Chenzhinan1219/article/details/122042407>

**附2：代码**

train.py:

import scipy.io as io

import matplotlib.pyplot as plt

import cv2

import time

import torch

import torch.nn as nn

import torchvision

import torchvision.transforms as transform

import torch.optim as optim

import numpy as np

from skimage import transform

import matplotlib.pyplot as plt

from ResNet import ResNet18

import os

import torch

import torchvision

from torch.utils.data import Dataset, DataLoader

from torchvision.transforms import transforms

from torchvision.utils import make\_grid

# 导入数据

data=io.loadmat('./data2/Dataset.mat')

# print(data.keys())

# print(data['train'].shape)

# print(data['test'].shape)

# plt.imshow(data['train'][0][0])

# 200类，每类15张训练集，5张测试集，图片size为28\*28

# 重写dataset类 加载自定义数据

class MyDataset(Dataset):

    def \_\_init\_\_(self,data,transform=None):

        self.data=data

        self.size=0

        self.transform=transform

        self.size=len(data)\*len(data[0])

#         print(len[data])

#         print(len(data[0]))

        self.name\_list=[]

        for i in range(0,len(data)):

            for j in range(0,len(data[0])):

                self.name\_list.append([i,j])

    def \_\_len\_\_(self): #\_\_len\_\_是指数据集长度。

        return self.size

    def \_\_getitem\_\_(self,idx): #\_\_getitem\_\_就是获取样本对，模型直接通过这一函数获得一对样本对{x:y}

        i=self.name\_list[idx][0]

        j=self.name\_list[idx][1]

        img=self.data[i][j]

        label=i

        sample={'image':img,'label':label}

        if self.transform:

            sample=self.transform(sample)

        return sample

def evaluate(model, dataloder):

    model.eval()

    correct = 0

    total = 0

    # since we're not training, we don't need to calculate the gradients for our outputs

    with torch.no\_grad():

        for data in dataloder:

            images, labels = data['image'],data['label']

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

            images=torch.reshape(images,(batch\_size,1,28,28))

            images=images.float()

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

            # calculate outputs by running images through the network

            outputs = model(images)

            # the class with the highest energy is what we choose as prediction

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

            total += labels.size(0)

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

    print('Accuracy of the network on the {} test images: {} %' .format(len(testloader),

            100 \* correct / total))

    return 100 \* correct / total

def draw\_loss(loss1,name):

    x=range(0,len(loss1))

    plt.title("loss")

    plt.xlabel("training times")

    plt.ylabel("loss")

    plt.plot(x,loss1)

    plt.legend() # 添加图例

    plt.savefig('./'+name+'.jpg')

    plt.show()

    plt.close()

def draw\_acc(acc,name):

    x=range(0,len(acc))

    plt.title("accuracy")

    plt.xlabel("training times")

    plt.ylabel("accuracy")

    plt.plot(x,acc)

    plt.legend() # 添加图例

    plt.savefig('./'+name+'.jpg')

    plt.show()

    plt.close()

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

            # 观察数据

    train\_dataset=MyDataset(data['train'],transform=None)

# print(train\_dataset.name\_list)

    test\_dataset=MyDataset(data['train'],transform=None)

    # 定义dataloader

    batch\_size=8

    trainloader = torch.utils.data.DataLoader(train\_dataset, batch\_size=batch\_size,shuffle=True, num\_workers=2)

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

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

    print(device)

# print(train\_dataset)

    plt.figure()

    for (cnt,i) in enumerate(train\_dataset):

        image = i['image']

        label = i['label']

#     print(image.shape)

        # ax = plt.subplot(3, 3, cnt+1)

        # ax.axis('off')

        # ax.imshow(image)

        # ax.set\_title('label {}'.format(label))

        # plt.pause(0.001)

        # if cnt == 8:

        #     break

    net= ResNet18(200)

    print(net)

    net.cuda()

    criterion = nn.CrossEntropyLoss()

    '''

    # try to change the learning rate

    '''

    optimizer1 = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

    optimizer2 = optim.SGD(net.parameters(), lr=0.0005, momentum=0.9)

    start = time.time()

    '''

    # try to change the number of total epoch

        '''

    k=0

    loss1=[]

    acc=[]

    for epoch in range(32):  # loop over the dataset multiple times

        net.train()

        running\_loss = 0.0

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

            # get the inputs; data is a list of [inputs, labels]

            inputs, labels = data['image'],data['label']

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

            # print(inputs.shape)

            inputs=torch.reshape(inputs,(batch\_size,1,28,28))

            inputs=inputs.float()

#             labels=labels.float()

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

            if k<300:

                optimizer=optimizer1

            else:

                optimizer=optimizer2

            # zero the parameter gradients

            optimizer.zero\_grad()

            # forward + backward + optimize

#             print(inputs.shape)

            outputs = net(inputs)

#             outputs=outputs.reshape(outputs.shape[0],1)

#             labels=labels.reshape(labels.shape[0],1)

#             print(outputs)

#             print(labels)

#             print(outputs.shape)

#             print(labels.shape)

#             print(type(outputs))

#             print(type(labels))

#             loss=nn.functional.cross\_entropy(outputs, labels)

            loss = criterion(outputs,labels)

#             print(loss)

            loss.backward()

            optimizer.step()

            # print statistics

            running\_loss += loss.item()

            #print(type(loss.item()))

            if i % 100 == 99:    # print every 2000 mini-batches

                print('[%d, %5d] loss: %.3f' %

                    (epoch + 1, i + 1, running\_loss / 100))

                loss1.append(running\_loss/100)

                running\_loss = 0.0

        # evaluate at the end of every epoch

        acc.append(evaluate(net, testloader))

    end = time.time()

    print('Finished Training {}s'.format(end-start))

    model\_name='resnet'

    PATH = './cifar\_net'+'model: '+model\_name+'.pth'

    torch.save(net.state\_dict(), PATH)

#draw\_loss(loss1)

    name1='loss'+'\_model:'+model\_name+'.npy'

    name2='acc'+'\_model:'+model\_name+'.npy'

    np.save(name1,loss1)

    np.save(name2,acc)

    name1='loss'+'\_model:'+model\_name

    name2='acc'+'\_model:'+model\_name

    loss1=np.load(name1+'.npy')

    print(loss1)

    acc=np.load(name2+'.npy')

    print(acc)

    draw\_loss(loss1,name1)

    draw\_acc(acc,name2)