

HW#5: Learning CNN

作业要求：

利用CNN进行手写数字识别与物体分类。

1. 实现最基本的卷机神经网络(CNN)LeNet-5以及一个物体分类的CNN，可以直接调用TensorFlow或PyTorch这2个常用的深度学习开发工具的各种构建函数。可直接调用开发工具的训练相关的接口，但不能直接读取各种深度学习开发工具已训练好的CNN网络结构与参数。
2. 自己用MNIST手写数字数据集(0-9一共十个数字)6万样本实现对LeNet-5的训练。对MNIST的一万测试样本进行测试，获得识别率是多少。
3. 自己用CIFAR-10数据库实现CNN物体分类功能的训练与测试。

个人信息：

姓名：鲍奕帆

学号：3180103499

邮箱：3180103499@zju.edu.cn



个人画像：

完成日期：2021年1月8日

一、软件开发说明

- 计算机与操作系统

Linux vipa-208 4.15.0-123-generic #126-Ubuntu SMP Wed Oct 21 09:40:11 UTC 2020 x86_64
x86_64 x86_64 GNU/Linux

CPU: Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz。

GPU:

```
(env2) byf@vipa-208:~/anaconda3/envs/env2/cifar$ lspci | grep -i nvidia
03:00.0 VGA compatible controller: NVIDIA Corporation GM200GL [Quadro M6000 24GB] (rev a1)
03:00.1 Audio device: NVIDIA Corporation GM200 High Definition Audio (rev a1)
04:00.0 VGA compatible controller: NVIDIA Corporation GP104GL [Quadro P5000] (rev a1)
04:00.1 Audio device: NVIDIA Corporation GP104 High Definition Audio Controller (rev a1)
```

内存: 80G

- 编程语言

python 3.6.2

- PyTorch版本

PyTorch 0.4.1

二、实验结果及展示

注: 考虑到以学习为目的, 节省时间, 因此这里的epoch次数比较小, 结果可能未收敛。

第一部分 LeNet-5 CNN手写字符识别

结果展示

训练过程以及结果

迭代次数为5次。batchsize=4。lr = 0.005。采用CrossEntropy作为loss, SGD优化器。

```
[5/5][49600/60000] loss: 0.028
[5/5][50000/60000] loss: 0.093
[5/5][50400/60000] loss: 0.072
[5/5][50800/60000] loss: 0.029
[5/5][51200/60000] loss: 0.031
[5/5][51600/60000] loss: 0.095
[5/5][52000/60000] loss: 0.028
[5/5][52400/60000] loss: 0.011
[5/5][52800/60000] loss: 0.038
[5/5][53200/60000] loss: 0.021
[5/5][53600/60000] loss: 0.064
[5/5][54000/60000] loss: 0.028
[5/5][54400/60000] loss: 0.059
[5/5][54800/60000] loss: 0.076
[5/5][55200/60000] loss: 0.046
[5/5][55600/60000] loss: 0.028
[5/5][56000/60000] loss: 0.055
[5/5][56400/60000] loss: 0.045
[5/5][56800/60000] loss: 0.029
[5/5][57200/60000] loss: 0.059
[5/5][57600/60000] loss: 0.071
[5/5][58000/60000] loss: 0.036
[5/5][58400/60000] loss: 0.005
[5/5][58800/60000] loss: 0.027
[5/5][59200/60000] loss: 0.045
[5/5][59600/60000] loss: 0.046
[5/5][60000/60000] loss: 0.032
model_5.pth saved
Finished Training
```

识别结果准确率

```
(env2) byf@vipa-208:~/anaconda3/envs/env2$ python test1.py  
correct rate is 98.790%
```

可以看到这个结果已经比较优秀了。

第二部分 CNN CIFAR-10 物体分类

自定义多层神经网络

迭代次数为4次。batchsize=4, lr=0.005.采用CrossEntropy损失函数, SGD优化器。

训练过程以及结果。

```
[4/4][48880/50000] loss: 0.728  
[4/4][48920/50000] loss: 0.647  
[4/4][48960/50000] loss: 0.915  
[4/4][49000/50000] loss: 0.773  
[4/4][49040/50000] loss: 0.597  
[4/4][49080/50000] loss: 0.522  
[4/4][49120/50000] loss: 0.957  
[4/4][49160/50000] loss: 1.020  
[4/4][49200/50000] loss: 0.703  
[4/4][49240/50000] loss: 0.642  
[4/4][49280/50000] loss: 0.703  
[4/4][49320/50000] loss: 0.755  
[4/4][49360/50000] loss: 0.549  
[4/4][49400/50000] loss: 1.192  
[4/4][49440/50000] loss: 0.782  
[4/4][49480/50000] loss: 0.805  
[4/4][49520/50000] loss: 1.026  
[4/4][49560/50000] loss: 0.793  
[4/4][49600/50000] loss: 0.645  
[4/4][49640/50000] loss: 0.396  
[4/4][49680/50000] loss: 0.747  
[4/4][49720/50000] loss: 1.172  
[4/4][49760/50000] loss: 0.884  
[4/4][49800/50000] loss: 0.941  
[4/4][49840/50000] loss: 0.929  
[4/4][49880/50000] loss: 0.688  
[4/4][49920/50000] loss: 0.494  
[4/4][49960/50000] loss: 0.705  
[4/4][50000/50000] loss: 0.724  
==> Saving model ...  
model_4.pth saved  
==> Finished Training ...
```

识别结果准确率

```
(env2) byf@vipa-208:~/anaconda3/envs/env2/cifar$ python test2.py  
correct rate is 71.090%
```

可以看到结果还不算太优秀，可能是自己设置的网络的问题，也可能是参数的问题，也可能是epoch次数的问题。

学习ResNet，利用ResNet进行训练。

ResNet第一次迭代结果测试

```
(env2) byf@vipa-208:~/anaconda3/envs/env2/cifar$ python test3.py  
correct rate is 51.62%
```

ResNet第二次迭代结果测试

```
(env2) byf@vipa-208:~/anaconda3/envs/env2/cifar$ python test3.py  
correct rate is 66.03%
```

ResNet第三次迭代结果测试

```
(env2) byf@vipa-208:~/anaconda3/envs/env2/cifar$ python test3.py  
correct rate is 71.53%
```

ResNet第四次迭代结果测试

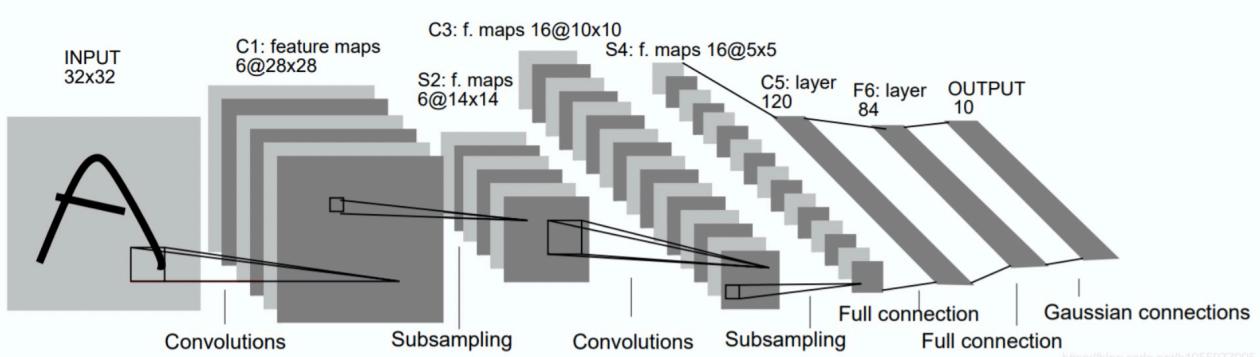
```
(env2) byf@vipa-208:~/anaconda3/envs/env2/cifar$ python test3.py  
correct rate is 75.63%
```

可以看到，使用ResNet的情况下，在第三次的结果就超过了自己定义的CNN。而第四次迭代结果性能又提升了很多。但由于时间限制，就没有进一步训练了。

三、算法步骤描述

第一部分 LeNet-5 CNN手写字符识别

1. Lenet-5介绍



LeNet中有2个卷积层，2个降采样(池化)层，以及3个全连接层(包括最后一个输出层)。

LeNet-5网络是针对灰度图进行训练的，输入图像大小为 $32 \times 32 \times 1$ 。输入图像的大小比MNIST数据集的图片要大一些，这么做的原因是希望潜在的明显特征如笔画断点或角能够出现在最高层特征检测子感受野 (receptive field) 的中心。

- 第一层是卷积层C1，这一层的输入就是原始图像的像素，输入图像为 $32 \times 32 \times 1$ ，卷积核大小为 5×5 ，通道数为6，步长为1，所以这一层的输出尺寸为 $32-5+1=28$ ，最终得到 $28 \times 28 \times 6$ 的tensor(即6个 28×28 的feature map)。C1中一共有156个可训练参数(每个滤波器大小为 5×5 ，同时有一个bias，一共有6个滤波器，因此总共参数个数为156)，有 $156 \times (28 \times 28)=122,304$ 个连接。同时采用ReLU函数作为激活函数。
- 第二层是一个下采样层S2，这一层使用 2×2 大小的过滤器，步长为2，所以这一层的输出尺寸为 $28 / 2 = 14$ ，最终得到 $14 \times 14 \times 6$ 的tensor。
- 第三层是卷积层C3，这里使用卷积核大小为 5×5 ，步长为1，通道数为16，最终得到 $10 \times 10 \times 16$ 的tensor。C3中一共有 $(5 \times 5 + 1) \times 16 = 416$ 个参数。同时采用ReLU函数作为激活函数。
- 第四层是下采样层S4，这一层使用的是 2×2 大小的过滤器，步长为2，最终得到的结果为 $5 \times 5 \times 16$ 的tensor

- 第五层是全连接层F5，本层输入向量维数为400，输出向量维数为120。一共有120个单元，每个单元与S4的全部400个单元之间进行全连接，一共有 $120 \times (400+1) = 48120$ 个可训练参数。激活函数采用ReLU
- 第六层是全连接层F6，本层输入向量维数为120，输出向量维数为84。一共有84个单元。每个单元与F5的全部120个单元进行全连接，一共有 $84 \times (120+1) = 10164$ 个可训练参数。激活函数采用ReLU
- 第七层是全连接输出层OUTPUT，本层输入向量维数为84，输出向量维数为10，将输入与输出进行全连接，最终输出是一个范围为[0,1]的10维向量，依次代表0-9。

2. 损失函数的定义

损失函数用来评价模型的预测值和真实值不一样的程度，从而判断模型的好坏。

常用损失函数的类型如下

- log对数损失函数

$$L(Y, P(Y|X)) = -\log P(Y|X)$$

log对数损失函数能非常好的表征概率分布，在很多场景尤其是多分类。

- 平方损失函数

$$L(Y|f(X)) = \sum_N (Y - f(X))^2$$

- 指数损失函数(exponential loss)

$$L(Y|f(X)) = \exp[-yf(x)]$$

- Hinge损失函数

$$L(y, f(x)) = \max(0, 1 - yf(x))$$

- 感知损失(perceptron loss)函数

$$L(y, f(x)) = \max(0, -f(x))$$

感知损失函数是Hinge损失函数的一个变种，Hinge loss对判定边界附近的点惩罚力度很高。而perceptron loss只要样本的判定类别正确的话就可以，不管其判定边界的距离。

- 交叉熵损失函数(Cross-entropy loss function)

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)]$$

$$loss = -\frac{1}{n} \sum_i y_i \ln a_i$$

其中 x 表示样本， y 表示实际的标签， a 表示预测的输出， n 表示样本总数量。交叉熵损失函数本质上也是一种对数似然函数，可用于二分类和多分类任务中。当使用sigmoid作为激活函数的时候，常用交叉熵损失函数而不用均方误差损失函数，因为它可以完美解决平方损失函数权重更新过慢点问题。函数是凸函数，求导时能够得到全局最优值。

交叉熵损失函数经常用于分类问题中，特别是在神经网络做分类问题时，也经常使用交叉熵作为损失函数。

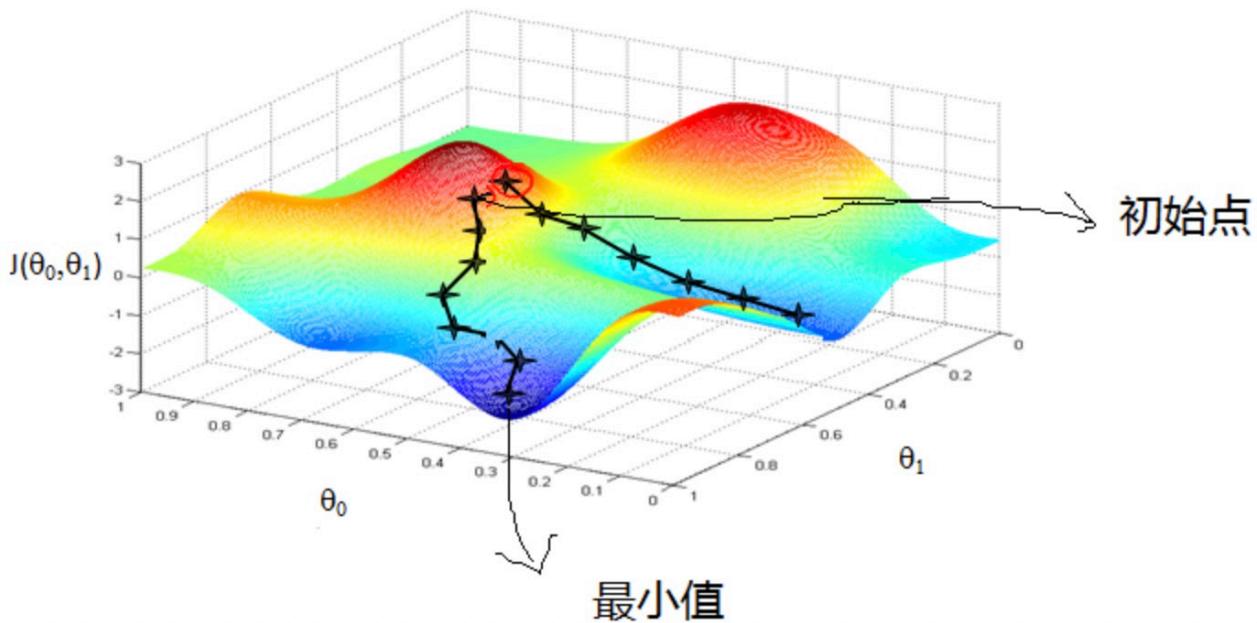
这也是本实验采用的损失函数。

3. 梯度下降算法原理

BP算法用于反向计算梯度，其利用的就是数学中的链式求导法则。

$$(f \circ g)'(x) = f'(g(x))g'(x)$$

多元函数的梯度方向是该函数值增大最陡的方向。本次实验采用随机梯度下降SGD方法。



代码描述

- 确认优化模型的假设函数和损失函数
- 算法相关参数初始化：主要是初始化 $\theta_0, \theta_1, \dots, \theta_n$ ，算法终止距离 e 以及步长 α 。
- 确定当前位置的损失函数梯度（利用BP算法求得-即forward再backward）
- 利用步长乘以损失函数的梯度，得到当前位置下降的距离
- 确定是否所有的 θ_i ，梯度下降的距离都小于 e ，如果小于 e 则算法终止，当所有的 $\theta_i (i=0, 1, \dots, n)$ 即为最终结果

- 更新所有的 $\theta_i = \theta_i - \alpha \frac{\partial}{\partial \theta_i} J(\theta_0, \theta_1 \dots, \theta_n)$

4. 本次实验完成步骤描述

- 使用torchvision加载和归一化MNIST训练和测试数据集
- 根据前面描述定义神经网络
- 定义损失函数和优化器
- 使用训练数据训练神经网络
- 在测试数据上测试网络性能

第二部分 构建CNN神经网络进行CIFAR-10数据集物体分类

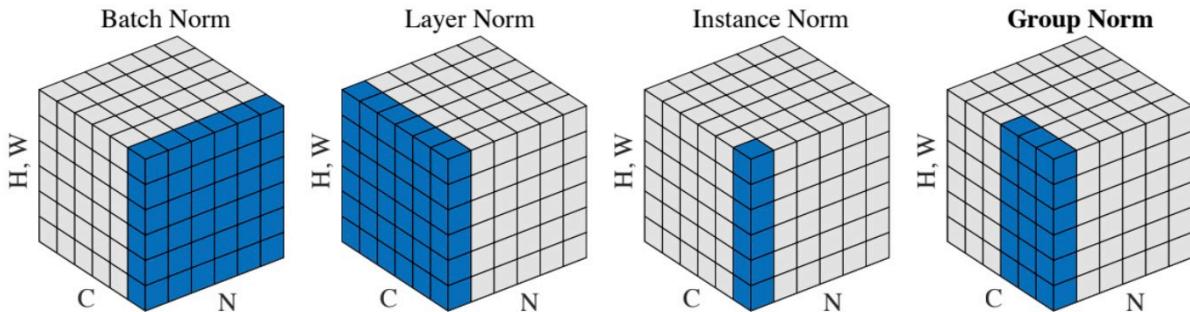
1. CIFAR-10数据集

与MNIST数据集相比， CIFAR-10有以下特点

- CIFAR-10是3通道的彩色RGB图像，而MNIST是单通道的灰度图像
- CIFAR-10的图像尺寸是32x32，而MNIST的图片尺寸为28 x 28
- CIFAR-10是现实世界中真实的物体，不仅噪声很大，而且物体的比例、特征都不尽相同。

2. 归一化层

归一化层包括BatchNorm、LayerNorm、InstanceNorm、GroupNorm、SwitchableNorm



记输入图像shape记为[N, C, H, W]

- BatchNorm:**

对NHW做归一化，对小batchsize效果不好。

算法过程

- 沿着通道计算每个batch的均值u
 - 沿着通道计算每个batch的方差 σ^2
 - 对x做归一化， $x' = (x - u) / \sqrt{\sigma^2 + \epsilon}$
 - 加入缩放和平移变量 γ 和 β ，归一化后的值， $y = \gamma x' + \beta$

尽管不同的Norm method有各自的特点，但本实验采用**BatchNorm**

- LayerNorm

对CHW做归一化，主要对RNN作用明显

- InstanceNorm

对HW做归一化，用在风格化迁移

- GroupNorm

将channel分组，然后在做归一化

- SwitchableNorm

将BN、LN、IN结合，赋予权重，让网络自己学习归一化层使用的方法

1. 自定义CNN网络架构

CIFAR-10输入图像为32x32x3

- 第一层为卷积层，输入通道为3，输出通道为32，卷积核为3，padding=1
- 经过BatchNorm以及ReLU函数激活
- 第二层是卷积层，输入通道为32，输出通道为64，卷积核大小为3，padding=1
- 经过Relu和MaxPooling
- 第三层是卷积层，输入通道为64，输出通道为128，卷积核大小为3，padding=1
- 经过BatchNorm和ReLU
- 第四层是卷积层，输入通道为128，输出通道为128，卷积核大小为3，padding=1
- 经过ReLU和MaxPooling和Dropout
- 第五层是卷积层，输入通道为128，输出通道为256，卷积核大小为3，padding=1
- 经过BatchNorm和ReLU
- 第六层是卷积层，输入通道是256，输出通道是256，卷积核大小为3，padding=1
- 经过ReLU和MaxPooling
- 第七层是全连接层，输入为4096的向量，输出为1024的向量。结果由ReLU激活
- 第八层是全连接层，输入为1024的向量，输出为512的向量。结果由ReLU激活，并且dropout
- 最后输出层也为全连接层，输入512的向量，输出10维的向量。

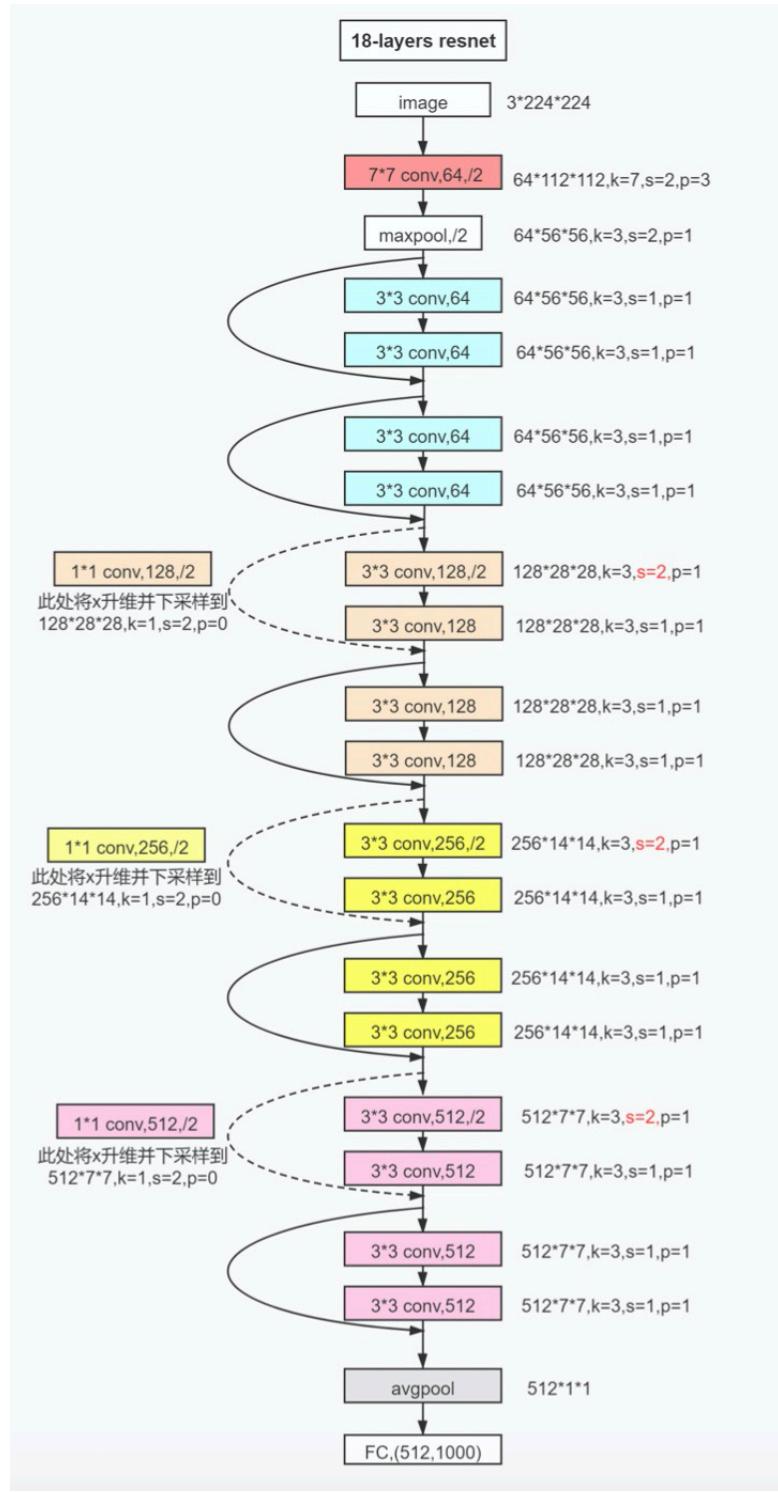
2. 本次实验完成步骤描述

- 使用torchvision加载和归一化CIFAR10训练和测试数据集
- 根据描述定义卷积神经网络
- 定义损失函数和优化器
- 使用训练数据训练神经网络
- 在测试数据上测试网络性能

3. 进一步尝试学习ResNet18进行CIFAR-10数据集物体分类

RestNet 网络结构

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112			7×7, 64, stride 2		
				3×3 max pool, stride 2		
conv2_x	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	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



四、算法具体实现

第一部分-Lenet-5 实现MNIST手写字符分类

1. Lenet网络定义 model1.py

```
import torch.nn as nn
import torch.nn.functional as F

# 标准LeNet-5
```

```

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 卷积层
        # @param1: input channels
        # @param2: output channels
        # @param3: kernel size
        self.conv1 = nn.Conv2d(1, 6, 5) # 输入为32 x 32 x 1的图像 输出为28 x 28 x
6的图像
        # 降采样后得到14 x 14 x 6的图像
        self.conv2 = nn.Conv2d(6, 16, 5) # 输入为14 x 14 x 6的图像 输出为10 x 10 x
16的图像
        # self.conv1 =
        nn.Conv2d(in_channels=1,out_channels=6,kernel_size=5,stride=1)
        # self.conv2 =
        nn.Conv2d(in_channels=6,out_channels=16,kernel_size=5,stride=1)
        # 降采样后得到5 x 5 x 16的图像

        # 全连接层
        # @param1: input vector dimensions
        # @param2: output vector dimensions
        self.fc1 = nn.Linear(16 * 5 * 5, 120) # 5 x 5 x 16 拉成vector 全连接输出得
到120维vector
        self.fc2 = nn.Linear(120, 84) # 120维vector全连接得到84维vector
        self.fc3 = nn.Linear(84, 10) # 84维vector全连接得到最终10维的输出结果

    def forward(self, x):
        # 卷积 --> ReLu激活函数 --> 池化(这里采用max pooling)
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
        # reshape, '-1'表示自适应
        # x = (n * 16 * 4 * 4) --> n : input channels
        # x.size()[0] == n --> input channels
        x = x.view(x.size()[0], -1)
        # 全连接层
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

    return x

```

上图为网络结构的定义，整体上各层基本上完全采用《Gradient-based learning applied to document recognition》中的结构。其中激活函数和池化层的选择是我自己定义的，也可以选择其他的方式。整体的数据输入也与我算法描述中的一致。

2. 训练部分代码 train1.py

```

import torchvision as tv
import torchvision.transforms as transforms

```

```
import torch
import torch.nn as nn
from torch import optim
from torch.utils.data import DataLoader
from torch.autograd import Variable
from model1 import Net
import numpy as np
if __name__ == '__main__':
    # 数据预处理-这里没有额外处理-就是转换为Tensor并且归一化至[0,1]
    transform = transforms.Compose([
        transforms.ToTensor(),
    ])
    # 训练集直接采用torchvision中的数据集 由于已经下载好了-这里download=False
    trainset = tv.datasets.MNIST(
        root='../../MNIST_data',
        train=True,
        download=False,
        transform=transform
    )
    trainloader = DataLoader(
        dataset=trainset,
        batch_size=4,
        shuffle=True
    )
    # MNIST数据集中的十种标签
    classes = ('0', '1', '2', '3', '4',
               '5', '6', '7', '8', '9')
# 创建网络模型
net = Net()
# 使用GPU
if torch.cuda.is_available():
    net.cuda()
# 采用交叉熵损失函数
criterion = nn.CrossEntropyLoss()
# 采用SGD优化器 学习率设置为0.005 动量设置为0.0
optimizer = optim.SGD(net.parameters(), lr=0.005, momentum=0.9)
# 迭代5个epoch
for epoch in range(5):
    running_loss = 0.0
    # 获取每一个batch的训练数据输入
    for i, data in enumerate(trainloader):
        # 输入数据
        inputs, labels = data
        inputs = inputs.numpy() # 转换为numpy
        # 得到的是1 x 28 x 28 x 1的numpy数组
        inputs = np.pad(inputs, ((0,0),(0,0),(2,2),(2,2)), 'constant') #
填充为32 x 32
        inputs = torch.tensor(inputs) # 转化回tensor方便之后处理
        # GPU
```

```

if torch.cuda.is_available():
    inputs = inputs.cuda()
    labels = labels.cuda()
inputs, labels = Variable(inputs), Variable(labels)
# 梯度清0
optimizer.zero_grad()
# forward
outputs = net(inputs)
# 计算loss
loss = criterion(outputs, labels)
# backward
loss.backward()
# 更新参数
optimizer.step()
# 打印日志信息
running_loss += loss.item()
# 每100个batch打印一次训练状态
if i % 100 == 99:
    print('[{}/{}][{}/{}]\t loss: {:.3f}'.format(epoch + 1, 5, (i + 1) * 4, len(trainset), running_loss / 100))
    running_loss = 0.0
# 保存模型
torch.save(net.state_dict(), 'checkpoints/model_{}.pth'.format(epoch + 1))
print('model_{}.pth saved'.format(epoch + 1))
print('Finished Training')

```

详细内容都已经在代码中，基本步骤与算法描述中的相同。包括了数据集的加载，超参数的定义，**epoch**的循环，输入的变量，以及训练过程中各种训练信息的输出。

3. 测试过程 test1.py

```

import torchvision as tv
import torchvision.transforms as transforms
import torch
from torch.utils.data import DataLoader
from torch.autograd import Variable
from model1 import Net
# 获取数据集
def get_dataset():
    # 数据预处理
    transform = transforms.Compose([
        transforms.ToTensor()
    ])
    # 测试集加载-也是已经下载好了
    testset = tv.datasets.MNIST(
        root='../../MNIST_data',
        train=False,
        download=False,

```

```
        transform=transform,
    )
testloader = DataLoader(
    dataset = testset,
    batch_size = 4,
    shuffle = True,
)
return len(testset), testloader
def main():
    # shou xie数字类别
    classes = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')
    # 初始化网络参数
    model = Net()
    # 加载模型
    model.load_state_dict(torch.load('checkpoints/model_5.pth'))
    # 使用GPU
    if torch.cuda.is_available():
        model.cuda()
    # 获取dataloader
    data_len, dataloader = get_dataset()
    # 正确预测的个数
    correct_num = 0
    for i, data in enumerate(dataloader):
        inputs, labels = data
        inputs = inputs.numpy() # 转换为numpy
        # 得到的是1 x 28 x 28 x 1的numpy数组
        inputs = np.pad(inputs, ((0,0),(0,0),(2,2),(2,2)), 'constant') # 填充为
32 x 32
        inputs = torch.tensor(inputs) # 转化回tensor方便之后处理
        # GPU
        if torch.cuda.is_available():
            inputs = inputs.cuda()
            labels = labels.cuda()
        inputs, labels = Variable(inputs), Variable(labels)
        outputs = model(inputs)
        # 取10维向量中的最大值为预测结果
        _, predicted = torch.max(outputs, 1)
        for j in range(len(predicted)):
            predicted_num = predicted[j].item()
            label_num = labels[j].item()
            # 预测值与标签值进行比较
            # 如果二者相等-则正确数加1
            if predicted_num == label_num:
                correct_num += 1
    # 最后计算准确率
    correct_rate = correct_num / data_len
    print('correct rate is {:.3f}%'.format(correct_rate * 100))
if __name__ == "__main__":
    main()
```

测试的过程包括 测试集的加载(这里也直接使用torchvision中下载的数据集), 遍历测试集合, 模型预测得到结果, 同时将预测结果与正确值进行比较, 如果二者相同, 则预测正确。最后输出识别率即可。

第二部分 利用CNN实现CIFAR-10上的物体分类

1. CNN网络模型定义 model2.py

```
import torch
import torch.nn as nn

class CNN(nn.Module):

    def __init__(self):
        super(CNN, self).__init__()
        self.conv_layer = nn.Sequential(
            # 第一块卷积
            nn.Conv2d(in_channels=3,out_channels=32,kernel_size=3,padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=32,out_channels=64,kernel_size=3,padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2,stride=2),

            # 第二块卷积
            nn.Conv2d(in_channels=64,out_channels=128,kernel_size=3,padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),

            nn.Conv2d(in_channels=128,out_channels=128,kernel_size=3,padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2,stride=2),
            nn.Dropout2d(p=0.05),

            # 第三块卷积
            nn.Conv2d(in_channels=128,out_channels=256,kernel_size=3,padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=256,
out_channels=256,kernel_size=3,padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2,stride=2),
        )

        self.fc_layer = nn.Sequential(
            nn.Dropout(p=0.1),
            nn.Linear(4096,1024),
            nn.ReLU(inplace=True),
```

```

        nn.Linear(1024,512),
        nn.ReLU(inplace=True),
        nn.Dropout(p=0.1),
        nn.Linear(512,10)
    )
    def forward(self, x):
        # 卷积部分
        x = self.conv_layer(x)
        # flatten 拉伸为vector
        x = x.view(x.size(0),-1)
        # 全连接部分
        x = self.fc_layer(x)
        #print("x in forward is")
        #print(x)
        return x

```

这里的设置与原理部分描述的内容完全一致。主要是学习了一些API的调用。

2. 训练过程 train2.py

```

import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim
from model2 import CNN
from torch.autograd import Variable
# 数据集加载
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    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=False, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True)

net = CNN()
# 使用GPU
if torch.cuda.is_available():
    net.cuda()
# 损失函数
criterion = nn.CrossEntropyLoss()
# 优化器
optimizer = optim.SGD(net.parameters(), lr=0.005, momentum=0.9)

```

```

for epoch in range(4):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        # GPU
        if torch.cuda.is_available():
            inputs = inputs.cuda()
            labels = labels.cuda()
        # 封装为Variable
        inputs, labels = Variable(inputs), Variable(labels)
        # 梯度清0
        optimizer.zero_grad()

        # forward
        outputs = net(inputs) #net
        # 计算损失函数
        loss = criterion(outputs, labels)
        # 反向传播
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        # 每10个batch打印一次训练状态
        if i % 10 == 9:
            print('[{}/{}][{}/{}]\tloss: {:.3f}'.format(epoch + 1, 4, (i + 1) * 4, len(trainset), running_loss / 10))
            running_loss = 0
        #if i % 100 == 99:
        #    print('[{}/{}][{}/{}]\tloss: {:.3f}'.format(epoch + 1, 5, (i + 1) * 4, len(trainset), running_loss / 100))
        #    running_loss = 0

    print('==> Saving model ...')
    state = {
        'net': net,
        'epoch': epoch,
    }

    torch.save(net.state_dict(), './checkpoint/model_{}.pth'.format(epoch+1))
    print('model_{}.pth saved'.format(epoch+1))

```

训练代码与第一部分MNIST手写字符识别基本相同，区别在于少了28x28到32x32图像的padding。参数与前面基本类似。

3. 测试过程 test2.py

```
import torchvision
```

```

import torchvision.transforms as transforms
import torch
from torch.utils.data import DataLoader
from torch.autograd import Variable
from model2 import CNN
# 数据集加载
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
testset = torchvision.datasets.CIFAR10(root='./data/', train=False,
download=False, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=False)

def main():
    classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck')
    model = CNN()
    model.load_state_dict(torch.load('checkpoint/model_4.pth'))
    # 使用GPU
    if torch.cuda.is_available():
        model.cuda()
    data_len = len(testset)
    correct_num = 0
    for i, data in enumerate(testloader):
        inputs, labels = data
        if torch.cuda.is_available():
            inputs = inputs.cuda()
            labels = labels.cuda()
        inputs, labels = Variable(inputs), Variable(labels)
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        for j in range(len(predicted)):
            predicted_num = predicted[j].item()
            label_num = labels[j].item()
            if predicted_num == label_num:
                correct_num += 1

    correct_rate = correct_num / data_len
    print('correct rate is {:.3f}%'.format(correct_rate * 100))

if __name__ == "__main__":
    main()

```

测试过程代码与前面也基本类似。区别在于class的名字，尽管这个并不影响。

最后给出ResNet的网络代码

```

import torch
import torch.nn as nn
import torch.nn.functional as F

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)
```

五、编程心得

本次实验难度适中。自己由于之前有在服务器安装过PyTorch以及相关的包，节省了较多的配置安装时间。然而自己对于神经网络的理解还是不够深入，尽管会写这类代码，会调整参数，但仍然感觉自己没有把握最本质，最精髓的部分。

同时，自己也没有做过多的调参内容，训练次数也不够多，因此第二部分的结果相对没有那么的可观。我认为自己还可以改进的地方是添加上tensorboard部分的内容，将训练时候的一些输出结果以可视化的形式展示出来，方便助教阅读。

总体上来看，收获还是比较大的，但对神经网络还是比较迷茫。其中迷茫的点在于：1. tensor的变化很难在脑中快速推演。2. 各个层运算后的结果也很难直接在大脑中计算得到。3. 对于各个层的作用还是不清晰。4. 对于各种优化方法仍然没有清晰的认识。

前路漫漫，仍需努力。

