华东师范大学计算机科学技术系实验报告

课程名称： 数据分析实践	**年级：** 2016 级	**实践作业成绩：**
指导教师： 兰曼	**姓名：** 张臻炜	**作业提交日期：**
实验编号： 4	**学号：** 10165102154	**实践作业编号：** 4

1 实验名称

手写体数字识别

2 实验目的

在前面数据分析的基础上，对手写数字图像进行预处理和构建识别系统。

通过手写体数字图像实例来进行数字图像的实践分析和识别分类，也适用于其他字符的多种图像类型数据。此外，本实验比较了基于不同分类算法的识别系统的性能。

实验要求

运用本学期的多种算法和策略进行手写体的数字识别。

3 实验内容

3.1 环境准备

3.1.1 基本环境

- Windows 10 专业版 x64

- Python 3.7.1
- pip 18.1

3.1.2　依赖

- scipy 1.2.0
- numpy 1.15.4
- matplotlib 3.0.2
- scikit-learn 0.20.2
- pytorch 1.1.0 (CPU version)

3.2　数据预处理

3.2.1　图像转为文本符号

在训练模型和进行预测之前，首先需要对图像做一些预处理工作，主要是将图片的二进制文件转变成文本文件，并进一步转化成特定维度的特征向量。

使用 PIL Python 图像库将图像转为文本符号。

Pillow 是 PIL 的一个友好分支 Fork，由 PIL 而来，支持 Python 3.x，导入该库使用 import PIL 即可，它提供非常广泛的文件格式支持，强大的图像处理能力，主要包括图像储存、图像显示、格式转换以及基本的图像处理操作等。

```
1 from PIL import Image, ImageDraw
2 import sys
3
4 # 将 256 灰度映射到 16 个字符上
5 def image_to_text(pixels, width, height):
6     symbols = list("$@B%8&WM#*oahkbdpqwmZOOQLCJUYXzcvu" +
7                    "nxrjft/\|()1{}[]?-_+~<>i!lI;:,\"^`'.")
```

```
    string = ""
    for h in range(height):
        for w in range(width):
            rgb = pixels[w, h]
            string += symbols[int(sum(rgb) / 3.0 / 256.0 * len(symbols))]
        string += "\n"
    return string

# 加载并调整大小
def load_and_resize_image(imgname, width, height):
    img = Image.open(imgname)
    if img.mode != 'RGB':
        img = img.convert('RGB')
    w, h  = img.size
    rw = width * 1.0 / w
    rh = height * 1.0  / h
    r = rw if rw < rh else rh
    rw = int(r * w)
    rh = int(r * h)
    img = img.resize((rw, rh), Image.ANTIALIAS)
    return img

# 图片转为文本
def image_file_to_text(img_file_path, dst_width, dst_height):
    img = load_and_resize_image(img_file_path, dst_width, dst_height)
    pixels = img.load()
    width, height = img.size
    string = image_to_text(pixels, width, height)
    return string
```

之后对于一张图像，调用 `image_file_to_text()` 函数即可。如图 1 为字符 `s` 转换成文本的表示、图 2 为小猪佩奇转换成文本的表示、图 3 和图 4 分别为字符 `2` 转换成文本的深色和浅色表示。

图 1: 字符 s

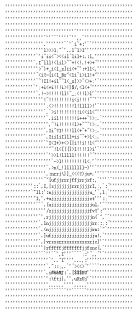

图 2: 小猪佩奇

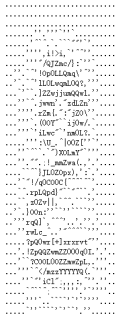

图 3: 字符 2 的表现形式 1

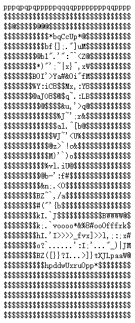

图 4: 字符 2 的表现形式 2

3.2.2 图像转为向量

首先将数据处理成分类器可以识别的文本格式，将 32×32 的二进制图像矩阵转换成 1×1024 的向量

```
def img2vect(filename):
    returnVect = np.zeros((1, 1024)) # 首先创建 1 E 1024 的 Numpy 数组
    fr = open(filename)
    for i in range(32):
        # 循环读出文件的前 32 行，并将每行的前 32 个字符值存储在 Numpy 数组中
        lineStr = fr.readline()
        for j in range(32):
            returnVect[0, 32 * i + j] = int(lineStr[j])
    return returnVect
```

3.2.3 获取训练和测试目录下的内容

因为目录下有很多文件，这些文件的命名都是有规律的，因此可以使用 `listdir` 来批量获取目录下的内容，并做处理。

```
from os import listdir
trainingFileList = listdir('./data/trainingDigits/') # 获取训练目录下的内容
print(" 训练样本个数: %d" % (len(trainingFileList))) # 训练数据样本个数
```

```
testFileList = listdir('./data/testDigits/') # 获取测试目录下的内容
print(" 测试样本个数：%d" % (len(testFileList))) # 训练数据样本个数
```

上面这段代码使用 listdir 获取了给定目录下所有文件的文件名称。

在获取了训练数据的文件名称之后，首先构造一个 1×1024 的空的矩阵，然后读取训练文件，并将文件中的 0 和 1 填入到这个矩阵中，同时，对文件名做 split 得到标签。

```
def load_trainingData():
    hwLabels = []
    trainingFileList = listdir('./data/trainingDigits/') # 获取训练目录下的内容
    m = len(trainingFileList) # 训练数据样本个数
    trainingMat = np.zeros((m, 1024)) # 矩阵每行存储一个图像
    for i in range(m):
        fileNameStr = trainingFileList[i]
        fileStr = fileNameStr.split('.')[0] # 从训练数据文件名解析分类数字
        classNumStr = int(fileStr.split('_')[0])
        hwLabels.append(classNumStr)
        trainingMat[i, :] = img2vect('./data/trainingDigits/%s' % fileNameStr)
    return trainingMat, hwLabels
```

测试数据集的加载是类似的。

```
def load_testData():
    testFileList = listdir('./data/testDigits')
    goldLabels = []
    mTest = len(testFileList)
    testMat = zeros((mTest, 1024))
    for i in range(mTest):
        fileNameStr = testFileList[i]
        fileStr = fileNameStr.split('.')[0]
        classNumStr = int(fileStr.split('_')[0])
        goldLabels.append(classNumStr)
```

```
        testMat[i, :] = img2vect('./data/testDigits/%s' % fileNameStr)
    return testMat, goldLabels
```

3.3 使用 kNN 进行手写数字识别系统的测试

3.3.1 kNN 算法简介

k 最近邻算法 目标函数可以是离散的，也可以是连续的，对于离散值，算法返回与预测样本最相似（即距离最近的）k 个训练样本的类标，对于连续纸，算法返回与预测样本最相似（即距离最近的）k 个训练样本的均指。

1. 所有的样本都对应是一个 n 维空间中的点，即，n 维的向量
2. 最近邻居定义为欧氏距离
3. 可以根据最近的 k 个邻居的距离进行加权，$w = \frac{1}{d(x_q, x_i)^2}$
4. kNN 算法具有噪音鲁棒性，即对噪音数据不敏感，因为只受最近的 k 个邻居影响

3.3.2 kNN 算法的实现

kNN 算法的实现比较简单，给定一个输入数据，在已知样本的集合中计算每一个点与给定输入点的距离，并将距离从小到大排序，取前 k 个点，即距离最小的 k 个点，统计这 k 个点的类标出现的个数，并以多数出现的类标作为给定输入的类标。

```
from numpy import tile, sum
import operator

#######################################
## 构建使用 kNN 的手写数字识别系统
#######################################

## kNN 算法，计算一个输入测试数据 inX 与已知样本的距离，并返回类别标签
def kNN(newInput, dataSet, labels, k):
```

```
    ## inX 是待分类测试样本，dataSet 是训练样本，labels 是训练样本标签
    dataSetSize = dataSet.shape[0] ## 行数，即样本个数

    ## step 1: calculate Euclidean distance
    # tile(A, reps): Construct an array by repeating A reps times
    # the following copy numSamples rows for dataSet
    diff = tile(newInput, (dataSetSize, 1)) - dataSet
    squaredDiff = diff ** 2
    squaredDist = sum(squaredDiff, axis = 1)
    distances = squaredDist ** 0.5

    ## step 2: sort the distance
    # argsort() returns the indices that would sort an array in a ascending order
    sortedDistIndicies = distances.argsort()

    classCount = {}# define a dictionary (can be append element)
    for i in range(k):
        ## step 3: choose the min k distance
        voteIlabel = labels[sortedDistIndicies[i]]
        ## step 4: count the times labels occur
        # when the key voteLabel is not in dictionary classCount, get()
        # will return 0
        classCount[voteIlabel] = classCount.get(voteIlabel,0) + 1
    ## step 5: the max voted class will return
    sortedClassCount = sorted(classCount.items(),
        key=operator.itemgetter(1),reverse=True)
    return sortedClassCount[0][0]
```

3.3.3　使用 kNN 进行手写数字识别系统的测试

加载训练数据和测试数据，对于每一个测试数据，使用 kNN 算法进行分类，并对结果进行比较。

```
def handwritingClassTest():
    hwLabels = []
```

```
    trainingFileList = listdir('./data/trainingDigits/') # 获取训练目录下的内容
    m = len(trainingFileList) # 训练数据样本个数
    trainingMat = np.zeros((m, 1024)) # 矩阵每行存储一个图像
    for i in range(m):
        fileNameStr = trainingFileList[i]
        fileStr = fileNameStr.split('.')[0] # 从训练数据文件名解析分类数字
        classNumStr = int(fileStr.split('_')[0])
        hwLabels.append(classNumStr)
        trainingMat[i, :] = img2vect('./data/trainingDigits/%s' % fileNameStr)
    # 解析测试数据文件
    testFileList = listdir('./data/testDigits/')
    errorCount = 0.0 # 统计识别错误的文件个数
    mTest = len(testFileList) # 测试数据样本个数
    for i in range(mTest):
        fileNameStr = testFileList[i]
        fileStr = fileNameStr.split('.')[0]
        classNumStr = int(fileStr.split('_')[0])
        vectorUnderTest = img2vect('./data/testDigits/%s' % fileNameStr)
        classifierResult = kNN(vectorUnderTest, trainingMat, hwLabels, 3)
        if (classifierResult != classNumStr):
            errorCount += 1.0
    print("\n 测试样本个数为: %d" % mTest)
    print(" 预测错误个数为: %d" % errorCount)
    print(" 预测错误率为: %2.2f%%" % (errorCount / float(mTest) * 100.0))
    print(" 预测准确率为: %2.2f%%" % ((1 - errorCount / float(mTest)) * 100.0))
```

3.4 比较几种分类算法对于手写体数字识别系统的的性能

首先导入不同分类算法的库。

```
from sklearn.neighbors import KNeighborsClassifier # SKlearn 中的 kNN 算法
from sklearn.svm import SVC # SKlearn 中的 SVC 算法
from sklearn.tree import DecisionTreeClassifier # SKlearn 中的决策树算法

# SKlearn 中的 NB 算法
```

```
from sklearn.naive_bayes import GaussianNB
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import BernoulliNB
```

然后依次尝试不同的分类器，调用 classifier.fit() 函数进行模型的训练、调用 classifier.predict() 进行预测，并比较不同分类算法的准确率。

```
# 基于几种不同分类算法的手写数字识别系统的代码
def handwritingClassTest():
    trainingMat, hwLabels = load_trainingData()
    testMat, goldLabels = load_testData()
    mTest = len(testMat) # 测试样本个数

    # 调用 sklearn 库中的分类算法
    ensemble = ["kNN", "SVC", "DT", "GaussianNB", "MultinomialNB", "BernoulliNB"]
    for a in ensemble:
        classifierResult = []
        print(a + ":")
        if a == 'kNN':
            clf = KNeighborsClassifier(algorithm='kd_tree', n_neighbors = 3)
        if a == 'SVC':
            clf = SVC(C = 1.0, kernel='linear')
        if a == 'DT':
            clf = DecisionTreeClassifier(criterion='entropy', random_state = 0)
        if a == 'GaussianNB':
            clf = GaussianNB()
        if a == 'MultinomialNB':
            clf = MultinomialNB()
        if a == 'BernolliNB':
            clf = BernolliNB()

        clf.fit(trainingMat, hwLabels) # 训练模型
        classifierResult = clf.predict(testMat) # 应用模型，预测测试数据

        errorCount = 0.0
```

```
29          for i in range(mTest):
30              if classifierResult[i] != goldLabels[i]:
31                  errorCount += 1.0
32
33          print("\n 测试样本个数为: %d" % mTest)
34          print(" 预测错误个数为: %d" % errorCount)
35          print(" 预测错误率为: %2.2f%%" % (errorCount / float(mTest) * 100.0))
36          print(" 预测准确率为: %2.2f%%" % ((1 - errorCount / float(mTest)) * 100.0))
```

3.5 手动实现 BP 神经网络进行手写体数字识别

上述各分类方法的准确率并不令人满意，下面将采用 BP 神经网络对手写体数字进行识别。

3.5.1 BP 神经网络简介

BP 神经网络是 1986 年由 Rumelhart 和 McClelland 为首的科学家提出的概念，是一种按照误差逆向传播算法训练的多层前馈神经网络，是目前应用最广泛的神经网络。

人工神经网络无需事先确定输入输出之间映射关系的数学方程，仅通过自身的训练，学习某种规则，在给定输入值时得到最接近期望输出值的结果。

BP 神经网络的基本思想是梯度下降法，利用梯度搜索技术，以期使网络的实际输出值和期望输出值的误差均方差为最小。

基本 BP 算法包括信号的前向传播和误差的反向传播两个过程。即计算误差输出时按从输入到输出的方向进行，而调整权值和阈值则从输出到输入的方向进行。

正向传播时，输入信号通过隐含层作用于输出节点，经过非线性变换，产生输出信号，若实际输出与期望输出不相符，则转入误差的反向传播过程。

误差反传是将输出误差通过隐含层向输入层逐层反传，并将误差分摊给各层所有单元，以从各层获得的误差信号作为调整各单元权值的依据。

通过调整输入节点与隐层节点的联接强度和隐层节点与输出节点的联接强度以及阈值，

使误差沿梯度方向下降，经过反复学习训练，确定与最小误差相对应的网络参数，训练即告停止。

3.5.2　实现神经元代码

神经元的数据域成员和方法比较简单，主要是记录一些信息，和完成一些特定的操作，相当于一个辅助类。

构造函数　神经元的构造函数需要接受神经元的数量、输入的数量，并且将各数组的元素初始化为 0。

```
# 默认构造函数，接受神经元数量，输入数量
def __init__(self, neuronNumbers, inputNumbers):
    self.neuronNumbers = neuronNumbers  # 神经元数量
    self.inputNumbers = inputNumbers  # 输入数量
    self.weight = []  # 权重
    for i in range(neuronNumbers):
        self.weight.append([])  # 对所有的神经元，初始化权重为 0
    self.outputActivations = []  # 输出层
    for i in range(neuronNumbers):
        self.outputActivations.append(0)  # 对所有的神经元，初始化输出层为 0
    self.errors = []  # 误差
    for i in range(neuronNumbers):
        self.errors.append(0)  # 对所有的神经元，初始化误差为 0
    self.reset()  # 重置
```

神经元的重置　神经元的重置就是将误差、输出层全部清空。为了达到更好的效果，我这里把权重的清空定义为：设置权重为随机值，而不是全 0，即，权重数组的每一个元素将在 [-0..5, 0.5] 区间内取值。

```
# 神经元重置
def reset(self):
```

```
3       # 清空误差、输出层数组，并对所有的神经元初始化误差、输出层为 0
4       self.errors.clear()
5       for i in range(self.neuronNumbers):
6           self.errors.append(0)
7       self.outputActivations.clear()
8       for i in range(self.neuronNumbers):
9           self.outputActivations.append(0)
10
11      # 对所有的神经元，设置随机的权重
12      for i in range(self.neuronNumbers):
13          weightNumbers = self.inputNumbers + 1  # 计算权重个数为输入个数 +1
14          self.weight[i].clear()
15          for j in range(weightNumbers):
16              self.weight[i].append(0)  # 清空权重数组，并全部初始化为 0
17
18          for j in range(weightNumbers):
19              self.weight[i][j] = random.random() - 0.5  # 随机 [-0.5,0.5) 的权重
```

3.5.3　BP 神经网络的实现

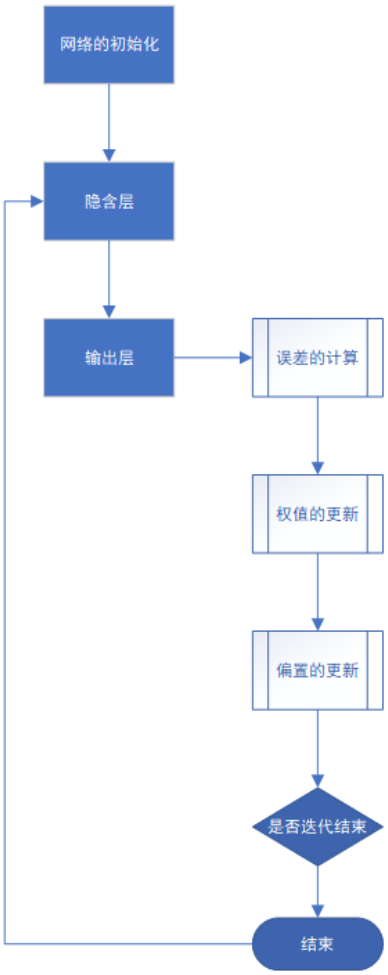

图 5: BP 神经网络的处理流程图

如图 5 为 BP 神经网络的处理流程，下面将逐步实现这一神经网络。

构造函数　构造函数中定义了一些全局的变量，包括学习率、隐藏层数量、偏移系数等。

前向传播　向前传播的函数，比较简单的是 S 型函数，即 Sigmoid 函数。

$S(x) = \frac{1}{1+e^{-x}}$

其中：x: 神经细胞所有输入（包括偏置输入）求和之后的值，$f(x)$: 神经细胞的最终输出

在实际的代码中，我给 x 增加了一个响应 RESPONSE。

RESPONSE 是用来对当前神经细胞所有输入信号（包括偏置）的累加和进行缩放的。这是因为 x 的取值负的越小，或者正的越大的时候，就越是逼近 0 或 1（饱和区）。如果在饱和区，由于反向传播是 Sigmoid 函数的导数，当逼近饱和区时，因为曲线变得平缓，斜率就趋近于 0。这样对于得到的权重变化就会非常非常小，也趋近于 0，也就是几乎达不到演化神经网络的目的（因为权重基本上没什么变化，就没有调整权重的意义了）。所以这里增加了一个响应 RESPONSE。

```
# 神经元重置
def sigmoid(self, x):
    return 1.0 / (1.0 + math.exp(-x * self.RESPONSE))
    # 响应 RESPONSE，主要用来修正 exp(x) 中 x 值过大过小可能导致所有的 sigmoid 值都为 1
```

反向传播 神经网络可能会有很多层，每一层有很多神经细胞，每一个神经细胞都有很多输入，与这些输入对应的就是很多权重。

首先得到网络的输出层的输出与我们期望之间的误差，然后再将这个误差值反向传播到前一层（隐含层），如此往复。最后就是根据误差，学习率以及当前取值点的梯度来更新每层神经细胞的输入权重，这就是所谓的反向传播算法。而不断调整神经细胞的权重以便输出误差达到最小的过程叫做梯度下降。

权重的调整 当前神经细胞的输出反向传播到了当前神经细胞的输入侧，而当前神经细胞的每一个输入，代表的都是反向传播方向的下一层神经细胞层的一个神经细胞。

$$\Delta W_{nj} = E_n LO_{(n-1)j}$$

在不同的 BP 神经网络中，对于 E 值的计算不尽相同，我在参考了众多参考资料后，决定使用下面这个公式计算 E 值。

$$E = \sum_{j=0}^{j=k} e_{(n+1)j} w_{(n+1)j}$$

对于输出层：E = 当前神经细胞的输出 - 预期输出。

对于隐含层：反向传播方向的上一层的所有神经细胞乘以与当前神经细胞的连接权重的累加和。

```
for j in range(inputNumbers):
    if oldErrors is not None:  # 输入层没有前一层
        # 更新前一层的误差，加和为当前的误差和权重的乘积
        oldErrors[j] = oldErrors[j] + error * weight[j]
    # 根据反向传播的规则更新误差
    weight[j] = weight[j] + error * self.learningRate * oldOutActivations[j]
# 偏移量
weight[inputNumbers] = weight[inputNumbers] + error * self.learningRate * self.BIAS
```

误差更新　训练过程会产生误差，所以常常需要重新去计算误差，然后更新误差。误差采用上文中 E 值的公式计算。

首先逐层计算。以输入层作为最初的输入，计算得到该层的输出，然后将前一层的输出作为下一层的输入，一层层地计算隐含层。取出最后一个隐含层的神经元，并获取它的输出和误差。

```
for i in range(self.hiddenNumbers + 1):
    self.updateNeuronLayer(self.layer[i], inputs)  # 用当前的输入，更新当前的神经元
    inputs = self.layer[i].getOutputActivations()  # 前一层的输出作为后一层的输入
```

下面重新计算误差。

对于每一个神经元，目标值减去输出值就是误差，将误差的平方累积起来。

误差采用 SSE，如果误差变小，说明训练是成功的。

```
for i in range(neuronNumbers):
    outErrors[i] = targets[i] - outActivations[i]
    self.errorSum = self.errorSum + outErrors[i] * outErrors[i]
```

训练并更新一层神经元　到这里，已经完成了正向和反向传播的函数，并完成了误差和权重的更新函数。那就可以开始训练神经元了。

首先遍历当前层的神经细胞，并计算每个神经细胞的输出误差以及调整权重的依据。在每一次遍历的过程中，利用反向传播激活函数计算反向传播回来的误差。

```
error = errors[i] * self.backActive(outActivations[i])
```

遍历当前神经细胞的所有权重，并基于反向传播回来的误差和学习率等参数计算新的权重值。

```
for j in range(inputNumbers):
    if oldErrors is not None:  # 输入层没有前一层，需要特别判断
        # 更新前一层的误差，加和为当前的误差和权重的乘积
        oldErrors[j] = oldErrors[j] + error * weight[j]
    # 根据反向传播的规则更新误差
    weight[j] = weight[j] + error * self.learningRate * oldOutActivations[j]
# 偏移量
weight[inputNumbers] = weight[inputNumbers] + error * self.learningRate * self.BIAS
```

对于每一个神经元，通过权重与输入的乘积的加和，作为 Sigmoid 函数的输入，获取响应输出，作为该神经元的输出。

遍历神经元的各个向量的时候，直接计算权重与输入的乘积并加和，最终把这个和作为 Sigmoid 函数的自变量，求出函数值，就是对应的输出。

定义训练和预测函数 下面，需要一个完整的训练函数，来整合正向的传播和反向的传播。

正向训练过程中顺便计算了误差，那么接下来就是要把误差反向传播，调整权重，以达到训练的目的。

下面以反向传播的顺序逐层训练网络。获取了第 i 层的神经元以后，需要做一个判断，那就是 i 是不是表示输入层。如果 i 是第 0 层，即这是输入层，那么它就是整个神经网络的输入，它没有前一层，于是，定义它前一层的输出就是整个神经网络的输入，同时，它没有前一层的误差。否则，就要取出它的前一层，并获取前一层的输出和误差。获取信息之后，调用神经元训练的函数对第 i 层神经元进行训练。

```
# 只有输入，没有前一层
oldOutActivations = inputs
oldErrors = None
```

```
prevNeuron = self.layer[i - 1]  # 前一层的神经元，获取其输出和误差
oldOutActivations = prevNeuron.getOutputActivations()
oldErrors = prevNeuron.getErrors()
```

预测部分相对简单，逐层计算，然后反馈，最终得到输出层。

```
# 预测，根据输入计算输出层
def predict(self, inputs):
    # 逐层计算
    for i in range(self.hiddenNumbers + 1):
        self.updateNeuronLayer(self.layer[i], inputs)  # 用当前的输入，更新当前的神经元
        inputs = self.layer[i].getOutputActivations()  # 前一层的输出作为后一层的输入
    # 得到输出层
    return self.layer[self.hiddenNumbers].getOutputActivations()
```

3.5.4　计算预测准确率

输出层是一个长度为 10 的数组，分别表示属于 0　9 各个数字的概率大小。很自然的，只需要遍历这个数组，找到值最大的，就是最终预测的答案，并将答案与标准答案进行比对。

```
index = -1
tmp = NEGATIVE_INFINITY
# 遍历，找到最大值，就是预测结果
for j in range(default_OutputLayerNumber):
    if out[j] > tmp:
        tmp = out[j]
        index = j
```

```
8
9  # 预测结果与测试答案比对，如果相等，说明预测正确，计数器加 1
10 if index == test_label[i]:
11     correctCount = correctCount + 1
12 # 输出预测结果和实际结果
13 print("Predict:", index, ",and the actual answer is:", test_label[i])
```

3.6 使用 pytorch 内置的神经网络进行手写体数字识别

准确率依旧不能令人满意，下面将尝试使用 pytorch 进行更多的神经网络训练。

首先，复现 *pytorch tutorial* 上对手写体数字的识别。

3.6.1 使用 pytorch tutorial 复现 MNIST 数据集的训练结果

https://pytorch.org/tutorials/intermediate/spatial_transformer_tutorial.html

代码冗长，见附录。

```
Test set: Average loss: 0.0448, Accuracy: 9871/10000 (99%)

Train Epoch: 18 [0/60000 (0%)]  Loss: 0.054946
Train Epoch: 18 [32000/60000 (53%)]     Loss: 0.084712

Test set: Average loss: 0.0391, Accuracy: 9880/10000 (99%)

Train Epoch: 19 [0/60000 (0%)]  Loss: 0.043331
Train Epoch: 19 [32000/60000 (53%)]     Loss: 0.266011

Test set: Average loss: 0.0416, Accuracy: 9885/10000 (99%)
```

图 6: 复现 pytorch tutorial 对手写体数字的识别

复现结果如图 6，准确率达 99%。

3.6.2　改写 pytorch 的 DataLoader 以训练我们自己的数据集

改写 DataLoader　创建一个针对自己数据集的 DataLoader，需要提供 3 个方法：

1. `def __init__ (self, loader = default_loader`: 初始化
2. `def __getitem__ (self, index)`: 在给定一个 index 的时候，返回一个图片的 tensor 和 target 的 tensor
3. `def __len__ (self)`: 所有数据的个数

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

import torch.utils.data as data
from os import listdir

def img2vect(filename):
    returnVect = np.zeros((32, 32)) # 首先创建 32 乘 32 的 Numpy 数组
    fr = open(filename)
    for i in range(32):
        lineStr = fr.readline()
        for j in range(32):
            returnVect[i, j] = int(lineStr[j])
    return returnVect

class load_trainingData(data.Dataset):
    trainingFileList = []
    m = 0

    def __init__(self):
        self.trainingFileList = listdir('./data/trainingDigits/')
        self.m = len(self.trainingFileList)

    def __getitem__(self, index):
        fileNameStr = self.trainingFileList[index]
        fileStr = fileNameStr.split('.')[0]
```

```
26          classNumStr = int(fileStr.split('_')[0])
27          target = torch.from_numpy(np.array(classNumStr))
28          data = torch.from_numpy(
29              np.array([img2vect('./data/trainingDigits/%s' % fileNameStr)]))
30          return data, target
31
32      def __len__(self):
33          return self.m
34
35  class load_testData(data.Dataset):
36      testFileList = []
37      mTest = 0
38
39      def __init__(self):
40          self.testFileList = listdir('./data/testDigits')
41          self.mTest = len(self.testFileList)
42
43      def __getitem__(self, index):
44          fileNameStr = self.testFileList[index]
45          fileStr = fileNameStr.split('.')[0]
46          classNumStr = int(fileStr.split('_')[0])
47          goldLabels = torch.from_numpy(np.array(classNumStr))
48          testMat = torch.from_numpy(
49              np.array([img2vect('./data/testDigits/%s' % fileNameStr)]))
50          return testMat, goldLabels
51
52      def __len__(self):
53          return self.mTest
54
55  # Training dataset
56  train_loader = torch.utils.data.DataLoader(load_trainingData(), batch_size=64, shuffle=
57  # Test dataset
58  test_loader = torch.utils.data.DataLoader(load_testData(), batch_size=64, shuffle=True)
```

数据必须是 tensor 格式的

将数据维度统一成 32 乘以 32 的大小

数据集总维度是 4 维，第一维度为 batch 的大小，我取 64，第二维度固定为 1，第三维度和第四维度为 32 乘以 32 的图片大小

数据格式统一

关于 Double 的问题，训练数据为 double 的，但是训练器是 float 的，有 2 个解决方案，一者是将训练网络转换成 double 的，另一种是将数据转换成 float 的，我采用了后者。

```
output = model(data.float())
```

如果修改神经网络，就是 `model = Net().double().to(device)`

矩阵相乘维度统一　进行修改的部分为：

```
self.fc1 = nn.Linear(500, 50)
self.fc2 = nn.Linear(50, 10)

xs = xs.view(-1, 160)

x = x.view(-1, 500)
```

4　实验结果及分析

4.1　使用 kNN 进行手写数字识别系统的测试

```
handwritingClassTest()
```

预测样本个数为 434 个，预测错误 17 个，准确率为 96.08%。

```
'''
测试样本个数为：434
预测错误个数为：17
预测错误率为：3.92%
预测准确率为：96.08%
'''
```

4.2 比较几种分类算法对于手写体数字识别系统的的性能

```
'''
kNN:
 测试样本个数为：434
 预测错误个数为：19
 预测错误率为：4.38%
 预测准确率为：95.62%
SVC:
 测试样本个数为：434
 预测错误个数为：18
 预测错误率为：4.15%
 预测准确率为：95.85%
DT:
 测试样本个数为：434
 预测错误个数为：76
 预测错误率为：17.51%
 预测准确率为：82.49%
GaussianNB:
 测试样本个数为：434
 预测错误个数为：120
 预测错误率为：27.65%
 预测准确率为：72.35%
```

```
MultinomialNB:
 测试样本个数为：434
 预测错误个数为：34
 预测错误率为：7.83%
 预测准确率为：92.17%
BernoulliNB:
 测试样本个数为：434
 预测错误个数为：34
 预测错误率为：7.83%
 预测准确率为：92.17%
'''
```

可以看到不同的分类器的准确率是不同的，并且相差挺大。

最好的分类器是 SCV 和 kNN，准确率接近 96%。

需要特别说明的是，由于取数据可能具有随机性（例如采用 k 折交叉验证来进行准确率评估的时候），以及部分算法的实现也带有随机因素，因此，多次运行上文代码可能会得到不同的结果。

4.3　手动实现 BP 神经网络进行手写体数字识别

```
0460385, 0.00012970946524742377, 0.000687723
 the actual answer is: 6
 which means the correct rate is: 93.91
```

图 7: BP 神经网络识别的准确率

如图 7，程序输出了预测结果以及真实结果，并对最终结果计算出准确率为 94%。

4.4 使用 pytorch 内置的神经网络进行手写体数字识别

```
Train Epoch: 13 [0/1498 (0%)]   Loss: 0.120983
pred= tensor(0) target= tensor(0) tensor(1, dtype=torch.uint8)
pred= tensor(7) target= tensor(7) tensor(1, dtype=torch.uint8)
pred= tensor(3) target= tensor(3) tensor(1, dtype=torch.uint8)
pred= tensor(1) target= tensor(1) tensor(1, dtype=torch.uint8)

Test set: Average loss: 0.1234, Accuracy: 422/434 (97%)

Train Epoch: 14 [0/1498 (0%)]   Loss: 0.083997
pred= tensor(4) target= tensor(4) tensor(1, dtype=torch.uint8)
pred= tensor(2) target= tensor(2) tensor(1, dtype=torch.uint8)
pred= tensor(1) target= tensor(1) tensor(1, dtype=torch.uint8)
pred= tensor(1) target= tensor(1) tensor(1, dtype=torch.uint8)

Test set: Average loss: 0.1235, Accuracy: 424/434 (98%)

Train Epoch: 15 [0/1498 (0%)]   Loss: 0.111689
```

图 8: 使用 pytorch 内置的神经网络进行手写体数字识别

如图 8，准确率稳定为 97% 98%。

历史最高准确率为 98%，在 434 个测试样本中，正确个数为 425 个。

5 问题讨论

修改上文 `load_testData` 中文件的路径。

由于测试数据没有标准标签，但是需要保留原始文件名以便最后输出预测结果，因此可以将 `target` 用作保存文件名。之后不再是将 `pred` 和 `target` 比较，而是直接输出 `target`（文件名）和 `pred` （预测结果）的一一对应关系。

```
1 test_loader = torch.utils.data.DataLoader(load_testData(), batch_size=64, shuffle=True)
2
3 f = open('torch.txt', 'w')
4 test_loss = 0
5 correct = 0
```

```
count = 0
for data, target in test_loader:
    data, target = data.to(device), target.to(device)
    output = model(data.float())

    pred = output.max(1, keepdim=True)[1]
    for i in range(len(pred)):
        print("file", target.long().view_as(pred)[i][0], "pred=", pred[i][0], file = f)
        count += 1

print(count, ' done!')
```

6　结论

历史最高准确率为 98%，在 434 个测试样本中，正确个数为 425 个。

6.1　感想

1. 聚类效果没有神经网络好。
2. 历史最高准确率为 98%，在 434 个测试样本中，正确个数为 425 个。
3. 导致准确率不能再上升的原因是由于训练样本太小，只有不到 2000 个，MNIST 完整数据有 6000 个，准确率可达 99% 以上。

复现 pytorch tutorial 的代码

```
from __future__ import print_function
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
```

```
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np

plt.ion()   # interactive mode

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Training dataset
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(root='.', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])), batch_size=64, shuffle=True, num_workers=4)
# Test dataset
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(root='.', train=False, transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])), batch_size=64, shuffle=True, num_workers=4)

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

        # Spatial transformer localization-network
        self.localization = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size=7),
            nn.MaxPool2d(2, stride=2),
            nn.ReLU(True),
            nn.Conv2d(8, 10, kernel_size=5),
```

```
            nn.MaxPool2d(2, stride=2),
            nn.ReLU(True)
        )

        # Regressor for the 3 * 2 affine matrix
        self.fc_loc = nn.Sequential(
            nn.Linear(10 * 3 * 3, 32),
            nn.ReLU(True),
            nn.Linear(32, 3 * 2)
        )

        # Initialize the weights/bias with identity transformation
        self.fc_loc[2].weight.data.zero_()
        self.fc_loc[2].bias.data.copy_(torch.tensor([1, 0, 0, 0, 1, 0], dtype=torch.flo

    # Spatial transformer network forward function
    def stn(self, x):
        xs = self.localization(x)
        xs = xs.view(-1, 10 * 3 * 3)
        theta = self.fc_loc(xs)
        theta = theta.view(-1, 2, 3)

        grid = F.affine_grid(theta, x.size())
        x = F.grid_sample(x, grid)

        return x

    def forward(self, x):
        # transform the input
        x = self.stn(x)

        # Perform the usual forward pass
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
```

```
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)


model = Net().to(device)

optimizer = optim.SGD(model.parameters(), lr=0.01)


def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        f = open('11.txt', 'w')
        print(len(data), file = f)
        print(len(data[0]), file = f)
        print(len(data[0][0]), file = f)
        print(len(data[0][0][0]), file = f)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 500 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
#
# A simple test procedure to measure STN the performances on MNIST.
#


def test():
    with torch.no_grad():
        model.eval()
        test_loss = 0
        correct = 0
```

```
118         for data, target in test_loader:
119             data, target = data.to(device), target.to(device)
120             output = model(data)
121 
122             # sum up batch loss
123             test_loss += F.nll_loss(output, target, size_average=False).item()
124             # get the index of the max log-probability
125             pred = output.max(1, keepdim=True)[1]
126             correct += pred.eq(target.view_as(pred)).sum().item()
127 
128         test_loss /= len(test_loader.dataset)
129         print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'
130               .format(test_loss, correct, len(test_loader.dataset),
131                       100. * correct / len(test_loader.dataset)))
132 
133 def convert_image_np(inp):
134     """Convert a Tensor to numpy image."""
135     inp = inp.numpy().transpose((1, 2, 0))
136     mean = np.array([0.485, 0.456, 0.406])
137     std = np.array([0.229, 0.224, 0.225])
138     inp = std * inp + mean
139     inp = np.clip(inp, 0, 1)
140     return inp
141 
142 # We want to visualize the output of the spatial transformers layer
143 # after the training, we visualize a batch of input images and
144 # the corresponding transformed batch using STN.
145 
146 
147 def visualize_stn():
148     with torch.no_grad():
149         # Get a batch of training data
150         data = next(iter(test_loader))[0].to(device)
151 
152         input_tensor = data.cpu()
153         transformed_input_tensor = model.stn(data).cpu()
154 
```

```
155         in_grid = convert_image_np(
156             torchvision.utils.make_grid(input_tensor))
157 
158         out_grid = convert_image_np(
159             torchvision.utils.make_grid(transformed_input_tensor))
160 
161         # Plot the results side-by-side
162         f, axarr = plt.subplots(1, 2)
163         axarr[0].imshow(in_grid)
164         axarr[0].set_title('Dataset Images')
165 
166         axarr[1].imshow(out_grid)
167         axarr[1].set_title('Transformed Images')
168 
169 for epoch in range(1, 20 + 1):
170     train(epoch)
171     test()
172 
173 # Visualize the STN transformation on some input batch
174 visualize_stn()
175 
176 plt.ioff()
177 plt.show()
```