机器学习导论作业报告

线性模型、神经网络、Adaboost 模型的实现与分析

姓名：李嘉琪　　学号：2018K8009915047　　专业：生命科学　　分组：1

一、数据集合与处理

- **MNIST 数据集**

1. 基本介绍

MNIST 是一个手写数字的数据库，有一个包含 60,000 个示例的训练集和一个包含 10,000 个示例的测试集。它是 NIST 中更大集合的子集。在固定大小的图像中，数字已被大小归一化并居中。每张图片为 28*28 像素，每个像素点的数值为 0-255 之间的整数。标签为 0-9 之间的整数。

2. 数据格式与读取

IDX 文件格式是用于各种数值类型的向量和多维矩阵的简单格式。读取该格式文件可采用 struct.unpack_from()函数。

```
import struct
def decode_idx3_ubyte(idx3_ubyte_file):
    bin_data = open(idx3_ubyte_file, 'rb').read()
    offset = 0
    magic_num, num_images, num_rows, num_cols = struct.unpack_from(fmt_header, bin_data, offset)
    print ('Read from the idx3 header---\nmagic number:%d, image number: %d, image size: %d*%d' %(magi
    image_size = num_cols * num_rows
    fmt_image = '>' + str(image_size) + 'B'
    offset += struct.calcsize(fmt_header)
    images = np.empty((num_images, num_rows, num_cols))
    images = np.empty((num_images, num_rows*num_cols))
    for i in range(num_images):
        images[i] = np.array(struct.unpack_from(fmt_image, bin_data, offset))
        offset += struct.calcsize(fmt_image)
    return images, num_images

def decode_idx1_ubyte(idx1_ubyte_file):
    bin_data = open(idx1_ubyte_file, 'rb').read()
    offset = 0
    fmt_header = '>ii'
    magic_num, num_images = struct.unpack_from(fmt_header, bin_data, offset)
    print("Read from the idx1 header---\nmagic number: %d, number of images: %d" %(magic_num, num_imag
    offset += struct.calcsize(fmt_header)
    fmt_image = '>B'
    labels = np.empty(num_images)
    for i in range(num_images):
        labels[i] = struct.unpack_from(fmt_image, bin_data, offset)[0]
        offset += struct.calcsize(fmt_image)
    return labels
```

3. 数据处理

取 28*28 = 784 个像素值作为数据特征，将特征归一化。在每个数据的最后一列加一维常数 1，作为线性组合中的常数项。因此经过处理的数据为 785 维、每一维取 0-1 之间小数的向量。使用 softmax 方法处理多分类问题之前，将标签化为 10 维、每一维取值 0 或 1、有且只有一维取 1 的向量。

- **SST2 数据集**

1. 基本介绍

stanford sentiment treebank 斯坦福情感树库是一个具有完全标记的解析树的语料库。每条数据是对电影的文字评论，对应的标签为 0 或 1，分别代表情感的负向和正向。SST2 提供三个经处理的数据集，分别为 train、test、dev，其中 test 不含标签。所以我们在实验中取 train 作为训练集，dev 作为测试集。

2. 数据格式与读取

下载到的数据为 tsv 格式，可以用 pandas 库进行读取。

```
import pandas as pd
path_dev = 'D:/Machine_learning/SST-2/dev.tsv'
path_train = 'D:/Machine_learning/SST-2/train.tsv'
path_test = 'D:/Machine_learning/SST-2/test.tsv'

def read_tsv(path):
    file = pd.read_csv(path,sep='\t')
    dict_file = file.to_dict()
    print("Have read the file as a dictionary.")
    return dict_file
```

3. 数据特征提取

采用词袋（Bag of words）模型进行特征提取。数据的每一维代表一个特定的词，这一维度的数字为这个词在这句话中出现的次数。维数为训练集中所有出现过的词的个数。我提取的数据共有 14789 维。

下面是具体的代码：

首先读取每个句子。

```
#%%
#Then read the sentences as lists, so that I know how to deal with it.
def read_sentences(dic):
    lis = []
    for i in range(len(dic['sentence'])):
        lis.append(dic['sentence'][i])
    print("Return a sentence list with %d elements."%(len(lis)))
    return lis

dev_list = read_sentences(dev_dic)
train_list = read_sentences(train_dic)
```

把句子分成单个的词，并把这些词组成词表。

```
#%%
#Split the sentences into words, and get rid of some simple punctuations.
#There are surely other things to remove, but I don't know how to do it except for working manually.
#So I did not make more changes.
def split_sentences(s):
    for i in range(len(s)):
        s[i] = s[i].replace('.','').replace(',','').split(' ')
    print("Have splited %d sentences."%(len(s)))
    return s

dev_split = split_sentences(dev_list)
train_split = split_sentences(train_list)
```

```
#%%
# Create a dictionary with all the words in the training set.
# But the returned dictionary is actually a list.
def create_my_dictionary(all_sentences):
    dictionary = ['The']
    for i in range(len(all_sentences)):
        for j in range(len(all_sentences[i])):
            c = 0
            for k in range(len(dictionary)):
                if all_sentences[i][j] == dictionary[k]:
                    break
                if all_sentences[i][j] != dictionary[k]:
                    c += 1
            if c == len(dictionary):
                dictionary.append(all_sentences[i][j])
        print("Checking words in the %dth sentence. ---Done"%(i))
    print("Have created a dictionary with %d words."%(len(dictionary)))
    return dictionary

dictionary = create_my_dictionary(train_split)
```

按照词表将每个句子化成向量。

```
# Look up every word in every sentence in the dictionary.
# Return data as vectors.
# This method is very slow.
def look_up_dic(dic,sp):
    """查字典，把句子化成向量"""
    ar = np.empty((len(sp),len(dic)),dtype = np.float16)
    for i in range(len(sp)):
        arr = np.zeros(len(dic))
        for j in range(len(sp[i])):
            k = 0
            for k in range(len(dic)):
                if sp[i][j] == dic[k]:
                    arr[k] += 1
        ar[i] = arr
        print("Have looked up the %dth sentence."%(i))
    print("All %d sentences done."%(len(sp)))
    return ar

dev_vector = look_up_dic(dictionary, dev_split)
train_vector = look_up_dic(dictionary, train_split)
```

```
#%%
# Read the labels.
def find_labels(dic):
    arr = np.empty(len(dic['label']))
    for i in range(len(dic['label'])):
        arr[i] = dic['label'][i]
    print("Have read %d labels."%(len(arr)))
    return arr

labels_dev = find_labels(dev_dic)
labels_train = find_labels(train_dic)
```

读取标签。归一化向量为 0-1 之间的值。

```
# Read the labels.
def find_labels(dic):
    arr = np.empty(len(dic['label']))
    for i in range(len(dic['label'])):
        arr[i] = dic['label'][i]
    print("Have read %d labels."%(len(arr)))
    return arr

labels_dev = find_labels(dev_dic)
labels_train = find_labels(train_dic)
#%%
# Normalize the vectors, so that every number is between 0 and 1.
# This function also add an additional dimention for every vector.
def normalize(x):
    x = np.true_divide(x,x.max(),)
    x = np.concatenate((x,np.ones((x.shape[0],1))),axis=1)
    print("The array has been normalized and added one external column. The shape is now",x.shape)
    return x

dev = normalize(dev_vector)
train = normalize(train_vector)
```

经过整理的数据，并在后面的测试中直接按照这些变量名调用：

```
"""经过整理的变量。这里所有x都经过了归一化，并且加上一列，作为线性组合的常数项。
各变量的维数为：
x_train_sst (67349,14790)
x_test_sst (872,14790)
y_train_sst (67349,)
y_test_sst (872,)
x_train_mnist (60000,785)
x_test_mnist (10000,785)
y_train_mnist (60000,)
y_test_mnist (10000,)
"""
```

二、学习模型与实现

- **线性分类模型**

Input: 训练数据集，训练标签，验证数据集，验证标签
超参数，包括梯度下降的步长（参数 a）、判断 Loss 收敛的 Threshold（参数 th）

Process:
1. 随机生成一组线性系数 w
2. 初始化变量 Loss_new，Loss_old，我分别设成 0 和 1
3. While { Loss_new 和 Loss_old 的绝对值之差大于 Threshold }, do
    a) Loss_old = Loss_new
    b) 利用训练数据集和训练标签进行一次梯度下降计算，更新系数 w
    c) 利用更新的 w 在验证数据集上计算 Loss_new
    End while

Output: 线性系数 w

以下函数是对上面过程的具体实现：

```python
def train_valid_softmax(x_train,x_validation,y_train,y_validation,a,k,threshold):
    """Linear model for multiple-class classification."""
    dx = x_train.shape[1]
    dy = y_train.shape[1]
    w = np.random.random((x_train.shape[1],y_train.shape[1]))
    l_new = 0
    l_old = 1
    c = 1

    while abs(l_new - l_old) >= threshold:
        l_old = l_new
        for i in range(k):
            t = random.randint(0, len(x_train)-1)
            p = fwx(x_train[t],w)
            w += a*np.dot(x_train[t].reshape(dx,1),(y_train[t]-p).reshape(1,dy))
        print("Have updated w %d times."%(c))
        l_new = error_multi(x_validation,y_validation,w)
        c += 1

    print("Finished learning. The min error on this validation set is %f"%(l_old))
    return w
```

实际应用中我还写了其他类似的函数，仅为了方便不同的调用做了一点点改动：

```python
def train_valid(x_train,x_validation,y_train,y_validation,a,k):
    """Linear model for binary classification."""
```

```python
def train_valid_softmax_dx(x_train,x_validation,y_train,y_validation,a,k,weightx,threshold):
    """Linear model for multiple-class classification with different sample weights."""
```

```python
def train_valid_softmax_nok_dx(x_train,x_validation,y_train,y_validation,a,k,weightx,threshold):
    """Linear model for multiple-class classification with different sample weights
    and it does not work with cross validation."""
```

```python
def train_valid_softmax_nok(x_train,x_validation,y_train,y_validation,a,k,threshold):
    """Linear model for multiple-class classification
    and it does not work with cross validation."""
```

- **前馈神经网络**

Input: 训练数据集，训练标签，验证数据集，验证标签
超参数，包括隐层的数目和隐层的维数（参数 hid）、梯度下降的更新步长（参数 a）

Process:
1. 随机生成两组线性系数 w, v
2. 初始化变量 Loss_new，Loss_old，我分别设成 0 和 1
3. While { Loss_new 和 Loss_old 的绝对值之差大于 Threshold }, do
    a) Loss_old = Loss_new
    b) 平方损失对 w 求偏导，以步长 a 更新 w
    c) 平方损失对 v 求偏导，以步长 a 更新 v
    d) 利用更新的 w 和 v 在验证数据集上计算 Loss_new
    End while

Output: 系数 w, v

具体实现如下：

```
#%%
def backpropagation(x_train,x_validation,y_train,y_validation,a,k,hid,threshold):
    """Grediant descent method for feedforward neural networks with one hidden layer."""
    y_train = softmax_y(y_train)
    y_validation = softmax_y(y_validation)
    v = np.random.random((x_train.shape[1],hid)) - np.random.random((x_train.shape[1],hid))
    w = np.random.random((hid,y_train.shape[1])) - np.random.random((hid,y_train.shape[1]))
    l_new = 1
    l_old = 0
    c = 1
    yd = len(y_train[0])
    xd = len(x_train[0])

    while abs(l_new - l_old) >= threshold:
        l_old = l_new
        for i in range(k):
            t = random.randint(0, len(x_train)-1)
            alpha = np.dot(x_train[t],v)
            h = 1/(1+np.exp(-alpha))
            pre = fwx(h,w)
            yty = np.dot(pre.reshape(1,yd),1-pre.reshape(yd,1))
            w -= a*yty*np.dot(h.reshape(hid,1),pre.reshape(1,yd)-y_train[t].reshape(1,yd))
            eph = -yty*np.dot(w,(pre.reshape(yd,1)-y_train[t].reshape(yd,1)))
            hth = np.dot(h.reshape(1,hid),1-h.reshape(hid,1))
            v += a*hth*np.dot(x_train[t].reshape(xd,1),eph.reshape(1,hid))

        print("Have updated w %d times."%(c))
        l_new = nn_error(x_validation,y_validation,v,w)
        c += 1

    print("Function backpropagation ---done. The min error on this validation set is %f"%(l_old))
    print("The error on the whole training set is %f."%(nn_error(x_train, y_train, v, w)))
    return w.v
```

类似函数:

```
def backpropagation_dx(x_train,x_validation,y_train,y_validation,a,k,hid,weightx,threshold):
    """Grediant descent method for feedforward neural networks with one hidden layer
    with different sample weigths."""
```

- **K-fold Validation**

Input: 训练数据集，训练标签
超参数 计划将训练集分成的份数 Part（P）
线性模型或神经网络模型

Process: 1. 将训练数据分为 P 份
2. For K = 1，2，…，Part，do
   a) 取第 k 个部分作为 Validation 集合，在剩下的 k-1 个部分上训练参数
   b) 输出训练得到的参数
   End for

Output: P 次得到的参数取算数平均（仅对线性模型成立，对其他模型应该采取投票的方式；仅仅由于线性性质，使得线性模型的参数平均和投票是等价的）

以下函数是对上面过程的具体实现：

```
def k_fold_multi(x,y,a,k,part,th):
    """k-fold function for multiple-class linear classification."""
    y_max = softmax_y(y)
    partition = len(x)//part
    w = []
    for i in range(part):
        x_validation = x[part:part+partition]
        y_validation = y_max[part:part+partition]
        x_train = np.concatenate((x[:part],x[(part+partition):]),axis = 0)
        y_train = np.concatenate((y_max[:part],y_max[(part+partition):]),axis = 0)
        w_k = train_valid_softmax(x_train,x_validation,y_train,y_validation,a,k,th)
        w.append(w_k)
        print("Finished the %dth fold_multi validation."%(i))

    w_all = np.zeros(w[0].shape)
    for i in range(len(w)):
        w_all += w[i]

    w_ave = w_all/part
    er_ave = error_multi(x, y_max, w_ave)
    print("Function k_fold_multi: done. Error on the whole training set is %f"%(er_ave))
    return w_ave
```

其他类似函数：

```
def k_fold(x,y,a,k,part):
    """k-fold function for binary linear classification."""
```

```
def k_fold_multi_dx(x,y,a,k,part,dx,th):
    """k-fold function for multiple-class linear classification
    with different sample weights."""
```

- **随机梯度下降 + MiniBatch**

Input: 训练数据集，训练标签，验证数据集，验证标签
待更新参数 w
超参数，包括梯度下降的步长（参数 a）、Mini Batch 集合的大小（参数 k）

Process: for i = 1, 2, ···, k, do
生成一个范围在训练数据集数据个数之内的随机整数 t
利用第 t 个训练数据和标签，计算梯度 gradient
W = w – a*gredient
End for

Output: 更新后的参数 w

具体的代码实现如下：

```
for i in range(k):
    t = random.randint(0, len(x_train)-1)
    p = fwx(x_train,w)
    w += weightx[t]*a*np.dot(x_train.reshape(dx,1),(y_train[t]-p).reshape(1,dy))
    """SGD for linear models."""
```

- **集成模型 Adaboost**

Input: 训练数据集，训练标签
获得简单机器学习模型的算法（我采用线性模型）
超参数：简单模型的个数 T

Process:
1. 初始化样本权重，各样本权重相等
2. 初始化 Adaboost 的预测结果 H = 0
3. For t = 1, 2, ···, T do
   a) 以考虑权重的样本学习得到一个线性模型
   b) 用该模型预测对训练集标签进行预测
   c) 计算这个模型的错误率 et
   d) 如果错误率 et < 0.5
      计算模型的权重 at
      根据模型的权重和对每个样本的预测正误情况，更新样本权重
      H += at*模型 t 的预测结果
   End for

Output: 集成预测结果 H

下面是对上面过程的具体实现（返回值为测试集上的错误率）：

```
def adaboost(x_tr, x_te, y_tr, y_tee, t, a, k, hid, th):
    """Adaboost with linear model as its simple classifiers."""
    y_te = softmax_y(y_tee)
    dx = np.ones((len(x_tr),1))
    alpha = np.zeros((t,1))
    hh = np.zeros((len(x_te),y_te.shape[1]))

    for i in range(t):
        w = train_valid_softmax_nok_dx(x_tr,x_tr, y_tr,y_tee, a, k,dx,th)
        er,yorn= error_multi_pp(x_tr, softmax_y(y_tr), w,dx)

        if er <= 0.5:
            alpha[i] = 0.5*np.log((1-er)/er)
            dx = dx_update(dx,alpha[i],yorn)
            hh += p_linear(x_te,y_te,w)*alpha[i]
            print("The %dth simple classifier has error rate %f"%(i,er))

    hh = ada_predict(hh)
    ea = sperror(hh,y_tee)
    print("Adaboost ---done. The error is %f"%(ea))
    return ea
```

类似函数:

```
def ada_plot(x_tr, x_te, y_tr, y_tee, t, a, k, hid, th):
    """Adaboost with linear model as its simple classifiers.
    It visualize ada's effect with plots."""
```

三、模型测试与参数调节

- **线性分类模型**

线性分类模型的超参数共有 4 个，分别为：

参数符号	参数名称	参数取值
a	T 梯度下降步长	待测试
k	Mini-batch 每次更新参数选择的集合大小	待测试

part	进行 k-fold Validation 时，将训练集合分成的份数，也是重复训练的次数	因为 MNIST 训练集和测试机的大小为 6：1，于是预先认为分成 6 份是最合适的
th	判断 Loss 收敛、停止循环的 threshold	0.001

- **在 MNIST 数据集上的测试**

1. 调节参数步长 a

刚刚开始调参，对于数量级甚至没有概念，所以先测数量级。

第一次粗调范围：

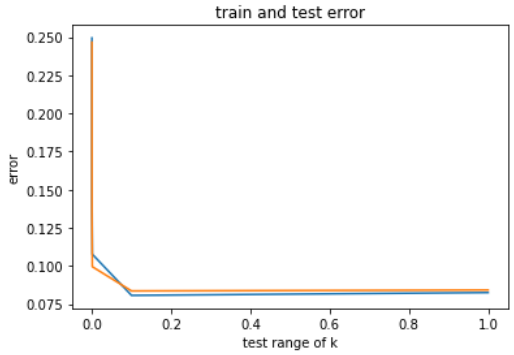
参数	a	k	part	th
取值	[1, 0.1, 0.001, 0.0001, 0.00001]	10000	6	0.001

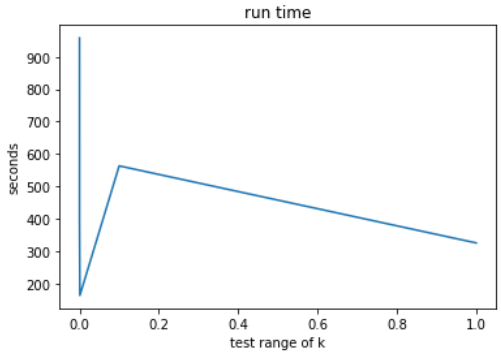
用于测试的函数：

```
def para_a(x_train,x_test,y_train,y_test,ran,part,k,hid,th):
    """hid参数nn才有"""
    tr = []
    te = []
    ti = []
    for a in ran:
        print("Running the a=%f test."%(a))
        start = datetime.now()
        w = k_fold_multi(x_train, y_train, a, k, part,th)
        train_error = error_multi(x_train,softmax_y(y_train),w)
        test_error = error_multi(x_test,softmax_y(y_test),w)
        #w,v = backpropagation(x_train, x_train, y_train, y_train, a, k, hid, th)
        #train_error = nn_error(x_train,softmax_y(y_train),v,w)
        #test_error = nn_error(x_test,softmax_y(y_test),v,w)
        run_time = datetime.now() - start
        tr.append(train_error)
        te.append(test_error)
        ti.append(run_time.seconds)

    plt.plot(ran,tr,label='train error')
    plt.plot(ran,te,label='test error')
    plt.xlabel('test range of a')
    plt.ylabel('error rate')
    plt.title('train and test error')
    plt.show()

    plt.plot(ran,ti)
    plt.xlabel('test range of a')
    plt.ylabel('seconds')
    plt.title('run time')
    plt.show()
```

中间注释的地方改一下也可以用来测试神经网络。后面测试其他参数也是用类似的函数框架。

测试结果（左图中蓝色为训练集上的错误率，黄色为测试集错误率，后面同样）：

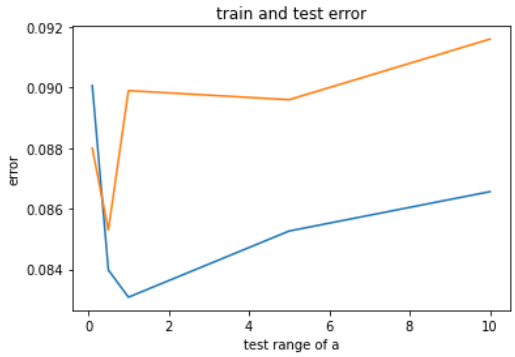


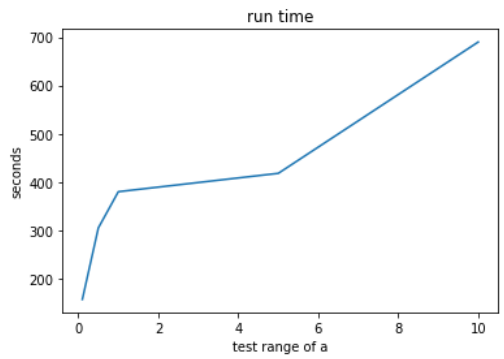


可见第一次测试范围可能取得不是很合适，步长应该再大一点。

第二次粗调范围

参数	a	k	part	th
取值	[10, 5, 1, 0.5, 0.1]	10000	6	0.001

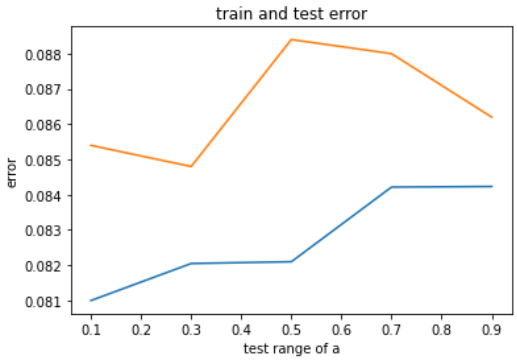


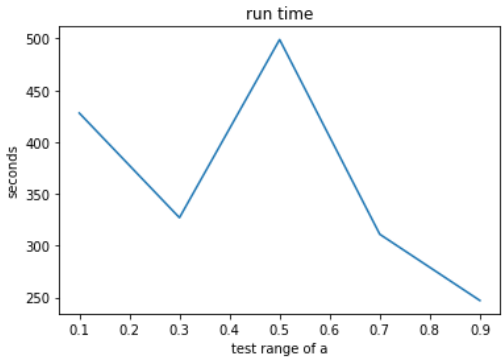


发现步长在 0.1 到 1 之间效果较好，时间也比较短。

第三次细调范围

参数	a	k	part	th
取值	np.arange(0.1, 1.1, 0.2)	10000	6	0.001

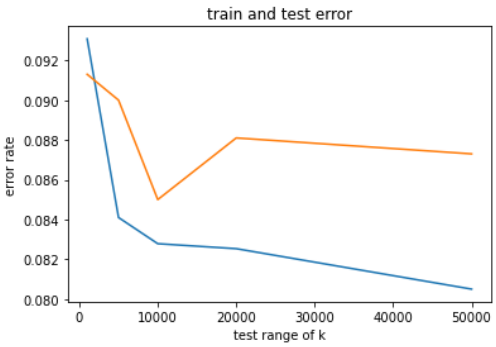


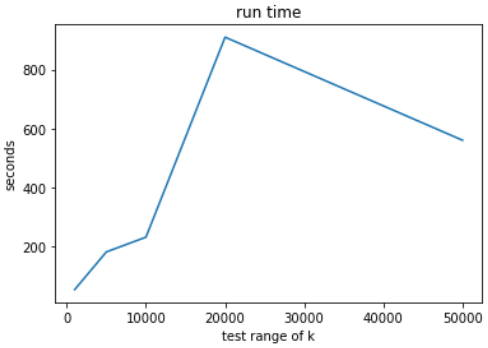


其实这个范围内取值都差别不大。

2. 调节参数 Mini-Batch 集合大小 k

参数	a	k	part	th
取值	1	[1000, 5000, 10000, 20000, 50000]	6	0.001

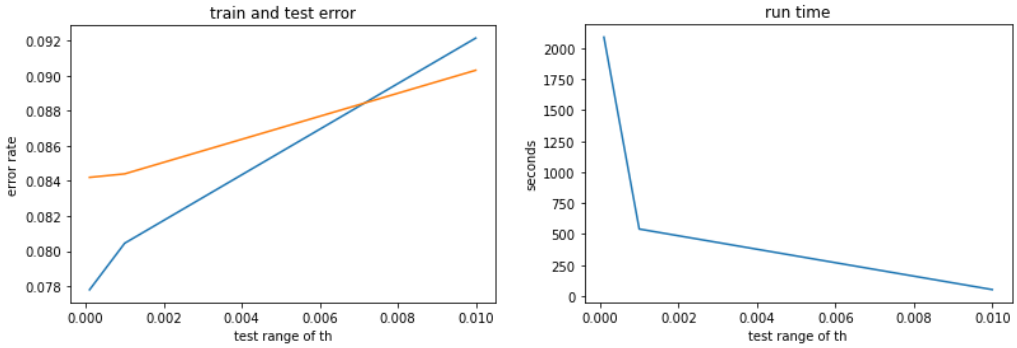




发现集合取得越大，训练集上的错误率越低，但是对测试集的错误并没有很大影响。而随着集合的增大，运行时间显著升高，这明显是不划算的。结果也显示一开始测试步长 a 时取的值 10000 确实是比较好的结果，后面可以继续用。也可以进一步缩小集合大小 k 进行测试，看看有什么影响。

3. 调节收敛判断条件 th

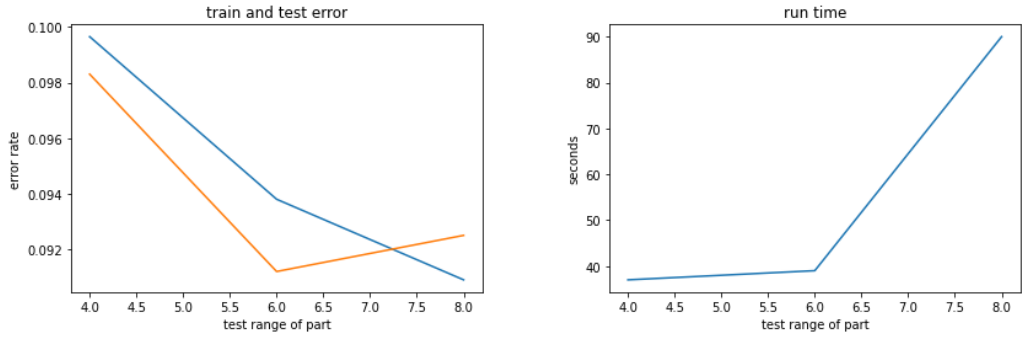
参数	a	k	part	th
取值	1	10000	6	[0.01, 0.001, 0.0001]



Threshold 取值越小，训练和测试的错误率越小，但是训练时间也快速增大。Threshold 从 0.001 讲到 0.0001 时，测试集上的进步已经很不明显，所以这时可能已经有过拟合的倾向了。Threshold 取 0.001 应该就足够了，想要运行更快的话，则取 0.01。这是稍稍有些欠拟合，但是影响不是很大。

4. 调节 k-fold 训练集合拆分个数 part

参数	a	k	part	th
取值	1	10000	[4, 6, 8]	0.01



拆分个数越多，训练时间越长，这是显然的；而拆分个数太少时，会造成每次用于训练的样本量太少，没法取得很好的效果。上面结果也显示，之前取的 part = 6 确实是比较合适的。

- **在 SST-2 数据集上的测试：**

调节参数步长 a

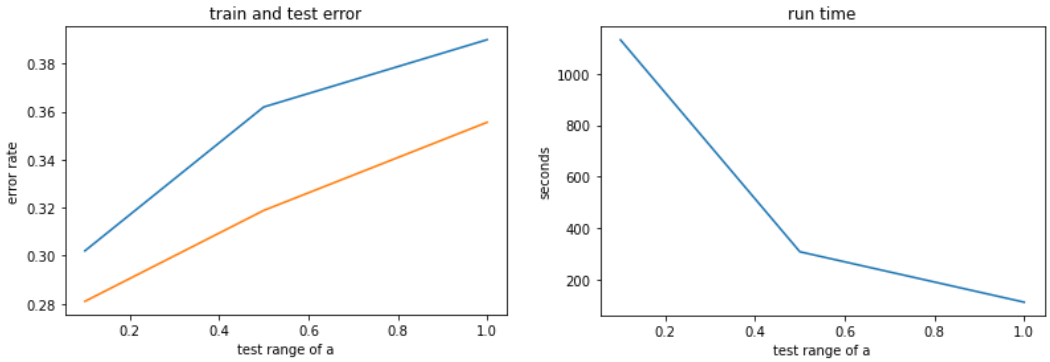
因为 SST-2 比较大，就不做多次的交叉验证了。线性模型的大多数参数应适用于不同的数据集。因为 SST-2 的维数比 MNIST 高很多，所以步长可能需要调整。其他参数采用上面优化的结果。步长测试范围如下：

参数	a	k	th
取值	[1, 0.5, 0.1]	10000	0.001

命令行：

```
ran = np.array([1,0.5,0.1])
para_al(x_train_sst,x_test_sst,y_train_sst,y_test_sst,ran,6,10000,1,0.001)
```

结果：



（1） 步长越大、训练时间越短，但是错误率也越高。因为 SST-2 维数比 MNIST 高一个数量级还多，适用的步长要小一个数量级，这也符合直觉。但是步长小的训练时间太长，测试受限。

（2） SST-2 的整体错误率在 0.3 左右，而 MNIST 大概在 0.1. SST-2 错误率高，可能是因为 SST-2 数据集本身作为文本的情感分析问题，没有 MNIST 易判断，而且我也没有进行什么有效的预处理；也可能是参数不适应。继续减小步长可能获得更小的错误率，但是时间成本太大。

- **前馈神经网络**

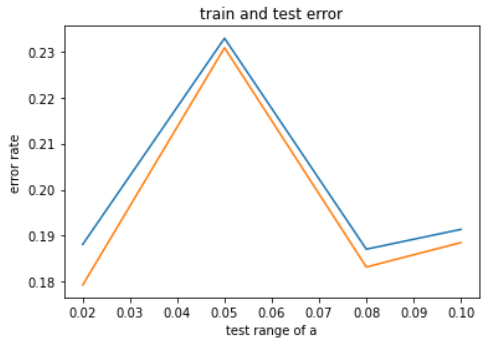
神经网络模型的超参数共有 5 个，分别为：

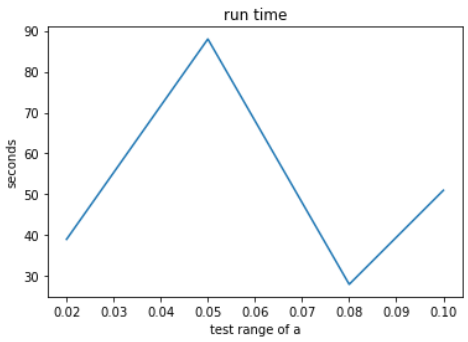
参数符号	参数名称	参数取值
a	T 梯度下降步长	待测试
k	Mini-batch 每次更新参数选择的集合大小	10000
th	判断 Loss 收敛、停止循环的 threshold	0.001/0.01
hid	隐层的维数	待测试

其中 Mini-batch 集合大小 k、训练集合划分数 part、以及收敛 threshold 对于线性模型和神经网络模型应该是可以共用的。所以固定这些参数，测试步长 a 和隐层维数 hid。

1. 调节步长 a

参数	a	hid	k	part	th
取值	[0.1,0.08,0.05,0.02]	10	10000	6	0.001

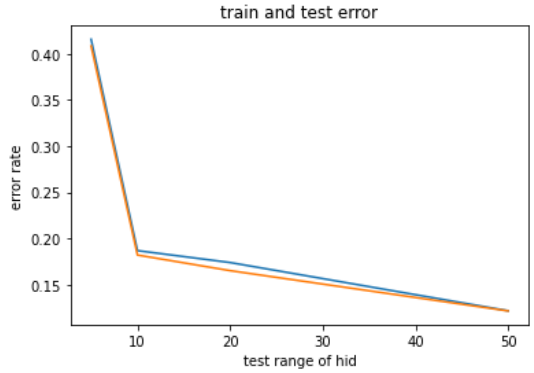


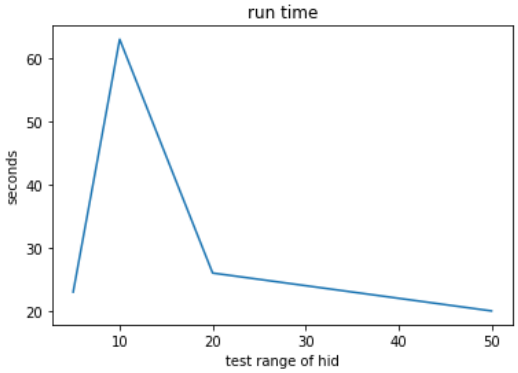


在测试的范围内，各个步长的表现差不多。后面测试暂且取 0.02，有空可以测试比 0.02 更小的步长，看看是否有本次测试范围未显示出的趋势性变化。

2. 调节隐层的维数 hid

参数	a	hid	k	part	th
取值	0.02	[5, 10, 20, 50]	10000	6	0.001





当维数增大时，错误率明显下降，运行时间也有所下降。感觉隐层的维数还可以再加一些，于是测试更大范围内的一组数据。

参数	a	hid	k	part	th
取值	0.02	[50, 100, 200, 500]	10000	6	0.001

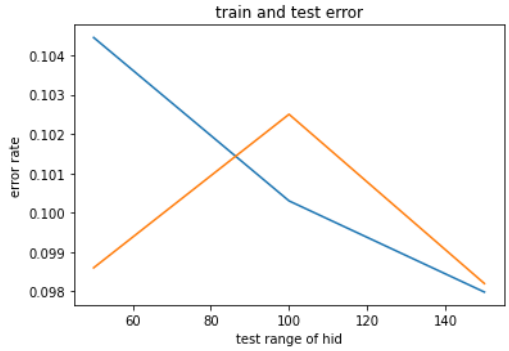
到了 200 就变得很慢了，500 维就没有算完，不过貌似也没有更低的错误率出现。

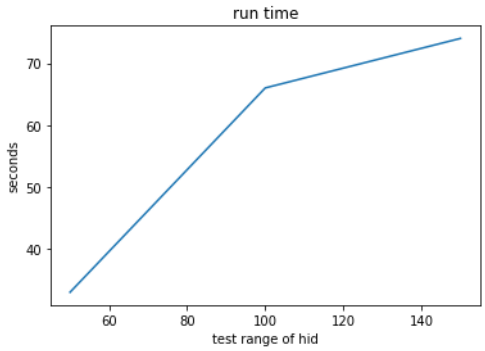
训练集错误率（50，100，200）：0.113483，0.102333，0.099950

测试集错误率（50，100，200）：0.109700，0.097600，0.102600

重新取一组数据画个图出来：

参数	a	hid	k	part	th
取值	0.02	[50, 100, 150]	10000	6	0.001

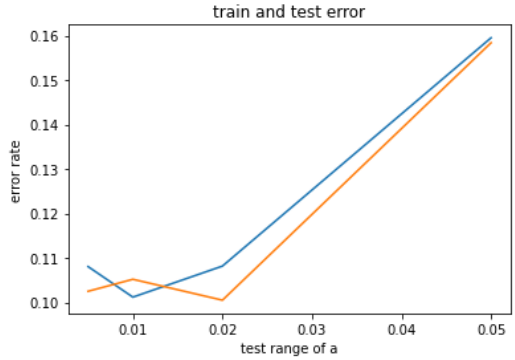


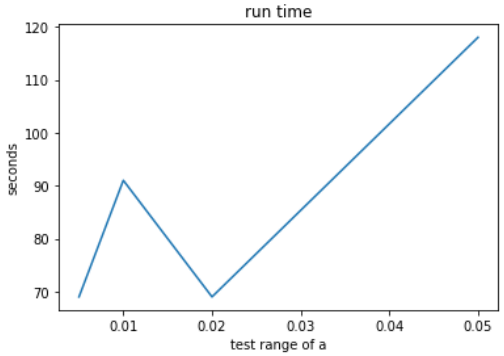


显然维数越高、运行时间越长。错误率并没有显著差异，因此 50 维应该是比较好的结果。

3. **对于更新后的维数，再次测试步长。**

参数	a	hid	k	part	th
取值	[0.05, 0.02, 0.01, 0.005]	50	10000	6	0.001





步长过大导致错误率升高，而小于 0.02 的步长表现区别不大。所以仍取步长为我们之前的 0.02. 这应该是神经网络模型比较优化的结果了。

➢ **在 SST-2 数据集上的测试**

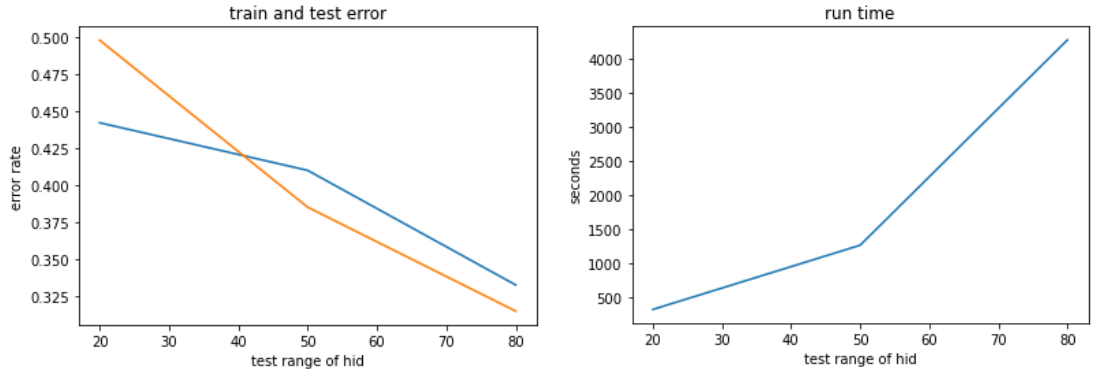
首先根据线性模型测试的结果，SST-2 的步长至少比 MNIST 小一个数量级，所以直接取 0.002 为步长，测试隐层维数对模型错误率的影响。在 MNIST 上，50 维的隐层已经有比较好的结果的，所以在 SST-2 上，取 50 维附近的值进行测试。

参数	a	hid	k	part	th
取值	0.002	[20, 50, 80]	10000	6	0.001

命令：

```
ran = np.array([20,50,80])
para_hid(x_train_sst,x_test_sst,y_train_sst,y_test_sst,0.002,6,10000,ran,0.001)
```

测试结果：



（1） 在这个参数的测试范围内，80 维隐层神经网络的表现还不如比较好的线性模型（步长 0.1），而训练时间高出 4 倍左右。假如不想花费更多时间训练模型的话（错误率允许比较高），线性模型在 SST-2 上的表现由于神经网络。

（2） 根据上面的变化趋势，有理由认为继续增加隐层维数能够降低错误率，但是时间实在太长了。于是我测试了 90 维。在一整天的训练之后（大概早上 7 点到晚上 8 点），神经网络在 SST-2 上给出了训练集 0.143684、测试集 0.214450 的错误率，相比 80 维有了明显的提升。

```
The nn_error rate is 0.214925
Have updated w 77 times.
The nn_error rate is 0.244488
Have updated w 78 times.
The nn_error rate is 0.144249
Have updated w 79 times.
The nn_error rate is 0.143684
Function backpropagation ---done. The min error on
this validation set is 0.144249
The nn_error rate is 0.143684
The error on the whole training set is 0.143684.
Have changed y into an array with shape (67349, 2)
The nn_error rate is 0.143684
Have changed y into an array with shape (872, 2)
The nn_error rate is 0.214450
```

- **集成模型 Adaboost**

1. 弱分类器

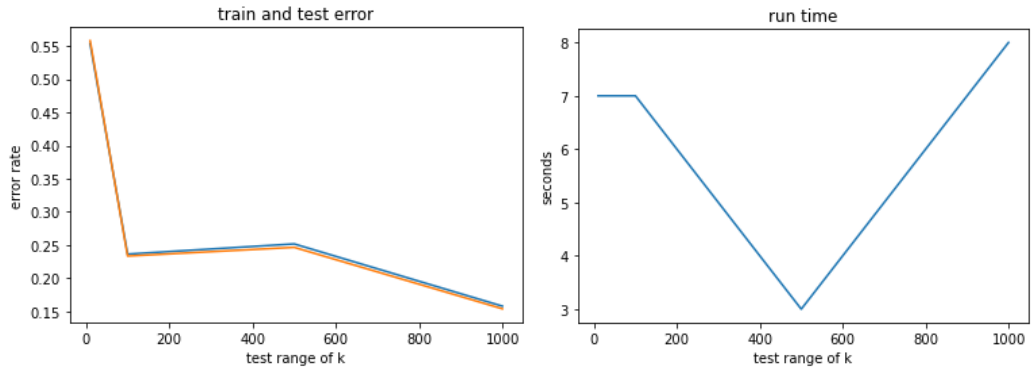
但是首先要获得一些弱分类器。考虑用线性模型作为 Adaboost 的弱分类器。（一开始觉得神经网络可能比线性模型慢很多，但其实并不，这个弱分类器的选择是主观的）

在前面的参数设置下，线性模型的错误率都在 0.1 以下，但是 Adaboost 其实只需要小于 0.5 的错误率就够了。前面这些线性模型的问题是速度太慢。于是考虑通过参数调节，获得一些速度更快、但是错误率允许比较高的线性模型。

首先取消 k-fold Validation，同时取 Threshold = 0.01，更快达到收敛条件。前面进行测试时发现 Mini-batch 集合大小对时间影响比较大，所以这里测试了比之前测过的 1000 更小的 k 值。

参数	a	k	th
取值	1	[10，100，500，1000]	0.01

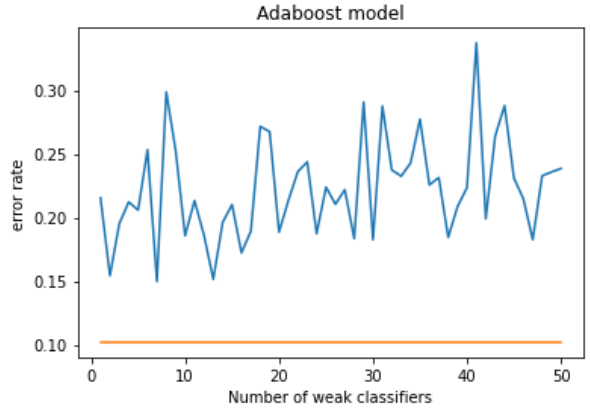
```
ran_kl = np.array([10,100,500,1000])
para_kl(x_train_mnist,x_test_mnist,y_train_mnist,y_test_mnist,1,6,ran_kl,1,0.01)
```



不妨取 k = 500，0.25 的错误率对于 Adaboost 来说是完全可以接受的。试试看 50 个这样的简单模型在 Adaboost 上的效果：

步长	Mini-batch 集合大小	弱分类器 Threshold	弱分类器个数	运行时间	Adaboost 错误率
1	500	0.01	50	291	0.1019

```
ada_plot(x_train_mnist, x_test_mnist, y_train_mnist, y_test_mnist, 50, 1, 500, 1, 0.01)
```

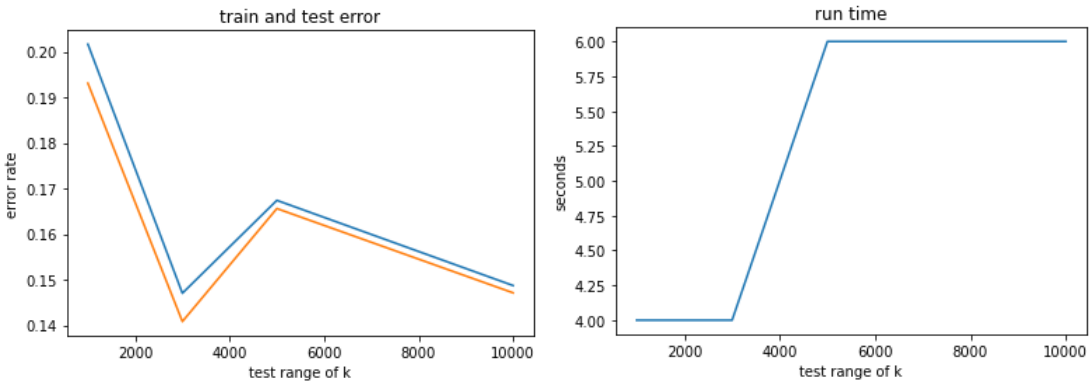


看到 Adaboost 确实做到了比各个弱分类器本身都好得多的效果。

再试一组弱分类器，这次希望把 th 取大一点，而 k 不必那么小。这样的基本结果就是，只在训练集中随机取出大小为 k 的子集进行训练。Threshold 取值较大的话，基本上就等于不存在，这就相当于挑合适大小的子集训练弱分类器。

参数	a	k	th
取值	1	[1000, 3000, 5000, 10000]	0.1

```
ran_kl = np.array([1000,3000,5000,10000])
para_kl(x_train_mnist,x_test_mnist,y_train_mnist,y_test_mnist,1,6,ran_kl,1,0.1)
```



发现对于 k = 3000, th = 0.1 也是一个不错的选择。

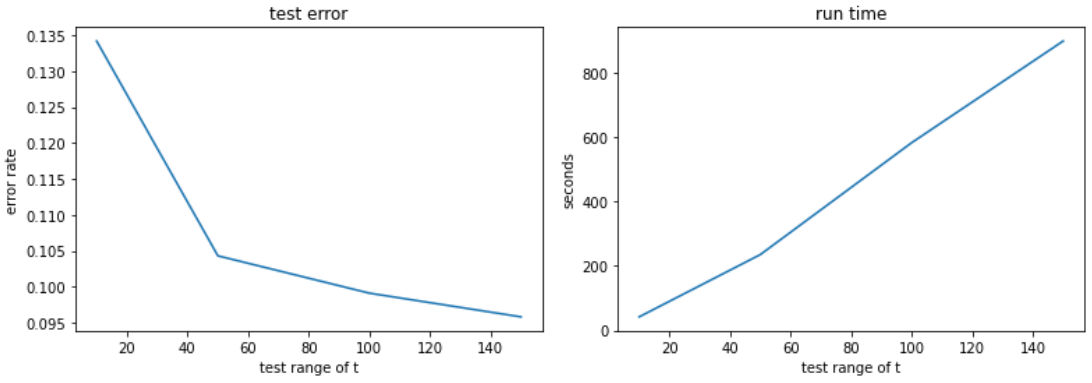
2. Adaboost 参数

Adaboost 的超参数只有一个：

参数符号	参数名称	测试范围
t	弱分类器个数	[10, 50, 100, 150]

下面用上一步调好的弱分类器调试 t：

```
ran = np.array([10,50,100,150])
para_ada(x_train_mnist,x_test_mnist,y_train_mnist,y_test_mnist,ran,1,500,50,0.01)
```



随着弱分类器个数的增长，显然运行时间基本呈线性增长。而分类的正确率依预期下降。按照 Adaboost 的原理，分类器越多，错误率越低，且不存在过拟合的状况。但是出于时间考虑，我决定用 50 个分类器就可以了。

四、模型比较与分析

1. 对线性模型、神经网络、Adaboost 三个模型的错误率、运行时间、稳定性比较

经过参数调节，最后对每个模型确定如下参数：

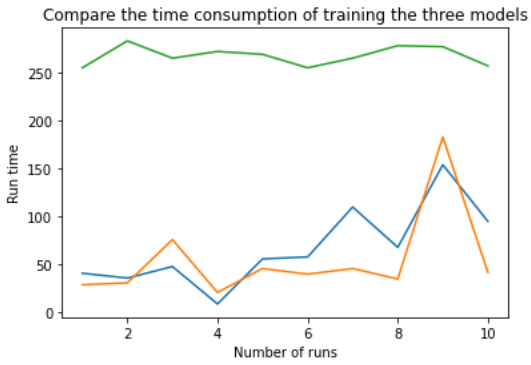
参数	a	k	part	hid	th	t
线性模型 k-fold	1	10000	6	-	0.001	-
线性模型	1	10000	-	-	0.001	-
神经网络模型	0.02	10000	-	50	0.001	-
Adaboost	1	500	-	-	0.01	50

运行如下代码：

```
def compare(x_tr,x_te,y_tr,y_te):
    tel = []
    ten = []
    tea = []
    tl = []
    tn = []
    ta = []
    for i in range(10):
        st = datetime.now()
        w = train_valid_softmax_nok(x_tr, x_tr, y_tr, y_tr, 1, 10000, 0.001)
        test_error = error_multi(x_te,softmax_y(y_te),w)
        tl.append((datetime.now()-st).seconds)
        tel.append(test_error)

        st = datetime.now()
        w,v = backpropagation(x_tr, x_tr, y_tr, y_tr, 0.02, 10000, 50, 0.001)
        test_error = nn_error(x_te,softmax_y(y_te),v,w)
        tn.append((datetime.now()-st).seconds)
        ten.append(test_error)

        start = datetime.now()
        ea = adaboost(x_tr, x_te, y_tr, y_te, 50, 1, 500, 50, 0.01)
        run_time = datetime.now() - start
        ta.append(run_time.seconds)
        tea.append(ea)

    plt.plot(np.arange(1,11),tel,label = 'linear model')
    plt.plot(np.arange(1,11),ten,label = 'neural network')
    plt.plot(np.arange(1,11),tea,label = 'Adaboost')
    plt.xlabel('Number of runs')
    plt.ylabel('Error rate')
    plt.title('Compare the error rates of the three models')
```

```
plt.show()

plt.plot(np.arange(1,11),tl,label = 'linear model')
plt.plot(np.arange(1,11),tn,label = 'neural network')
plt.plot(np.arange(1,11),ta,label = 'Adaboost')
plt.xlabel('Number of runs')
plt.ylabel('Run time')
plt.title('Compare the time consumption of training the three models')
plt.show()
```

运行得到的结果如下（蓝色为线性模型，黄色为神经网络，绿色为集成模型）：



（1） 线性模型的错误率高于神经网络和集成模型，后二者的错误率不相上下。线性模型错误率在 0.12 左右，而神经网络和集成模型的错误率都在 0.10 左右。

（2） 三个程序的运行时间都不算短，一方面可能与我的代码相关，另一方面与电脑性能相关。后面运行时间出现上升趋势，应该是因为我同时在电脑上开了别的程序。

（3） 由于我的模型中都有取随机数的部分，所以专门运行了多次检测稳定性。线性模型和神经网络的在不同的测试中，错误率和运行时间都有相对较大的波动，而集成模型比较稳定。

（4） 其实集成模型应该是很好的算法，它的错误率和稳定性都很好。我这里唯一的问题是运行太慢。运行慢的原因应该源于我的弱分类器太慢。在维数比较高的情况下，线性模型没办法很快，所以可能本身不很适合做集成模型的弱分类器。

2. 对线性模型、神经网络、Adaboost 三个模型所需的训练数据量比较

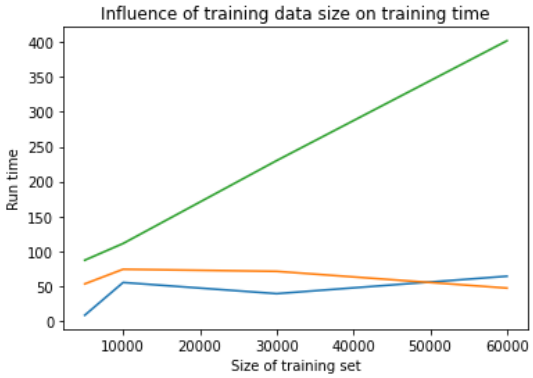
使用 MNIST 数据集，共有 60000 个训练数据，测试训练数据量对于三个模型的影响。测试的集合大小范围是 [5000, 10000, 30000, 60000]。

测试命令：

```
amount_data(x_train_mnist, x_test_mnist, y_train_mnist, y_test_mnist)
```

其中 amount_data（）函数结构和前面的 compare（）函数类似。

测试结果如下（蓝色为线性模型，黄色为神经网络，绿色为集成模型）：



（1） 线性模型的错误率仍然比神经网络和集成模型高，这与前面的测试结果一致。

（2） 随着训练集合增大，线性模型在测试集上的错误率下降明显。结合训练过程中的输出，这是因为线性模型在比较小的集合上容易出现过拟合。而在大的训练集上，它再怎么过拟合，总是无法兼顾到全部数据；神经网络和集成模型的错误率对训练集大小完全基本不敏感。我的测试的最小集合已经取到 5000 了，测试结果还是很不错。

（3） 随着训练集合增大，集成模型的学习时间几乎是线性增加，而线性模型和神经网络的训练时间也对集合大小不敏感。这应该是由于我线性模型和神经网络都是采用 Mini-batch + SGD 方法。这个结果也体现了 Mini-batch 的优势。

（4） 总之，对于训练数据较少的情况，神经网络网络和集成模型相较线性模型都有很大优势，而随着训练集的增大，神经网络模型又比集成模型具有更大优势。不过，不计成本的话，集成模型的错误率可以不断下降，神经网络似乎没办法做到。

3. 总结

从上面有限的测试结果中，三个模型呈现以下特点，这可能和我的具体实现也有关：

模型	错误率	训练时间	稳定性	需要的训练集大小
线性模型	相对最高	相对较短	相对较差	需要比较大的训练集来避免过拟合（如果没有其他避免过拟合的手段）
神经网络	较低	相对较短	相对较差	较小，且错误率对训练集合大小不敏感
集成模型	较低，而且随着弱分类器个数和训练集大小的增加，可以不断降低	和弱分类器的产生速度相关。随训练样本量、弱分类器个数的增加线性增大	较好	较小，但是随着训练集增加，错误率可以降低

五、遇到的问题和解决

1. 神经网络出现迭代提前终止

一开始允许神经网络时，出现这样的问题：

运行命令

```
backpropagation(x_train_mnist,x_train_mnist,y_train_mnist,y_train_mnist,0.02,10000,50,0.001)
```

结果

```
In [23]: runcell(33, 'D:/Machine_learning/sst2.py')
Have changed y into an array with shape (60000, 10)
Have changed y into an array with shape (60000, 10)
Have updated w 1 times.
The nn_error rate is 0.901283
Have updated w 2 times.
The nn_error rate is 0.901283
Function backpropagation ---done. The min error on this
validation set is 0.901283
The nn_error rate is 0.901283
The error on the whole training set is 0.901283.
```

可见系数迭代仅进行了一次就停止了，因为错误率的改变为 0，直接达到收敛条件，而这时模型显然还没有进行学习（这里测试的是 MNIST 数据集，有 10 个类，错误率 0.9 就是随机预测的结果）。考虑这应该是更新系数时，更新量中某一项为零。

这是更新系数部分的代码（红框标出更新项，这一部分等于零导致迭代终止）：

```
while abs(l_new - l_old) >= threshold:
    l_old = l_new
    for i in range(k):
        t = random.randint(0, len(x_train)-1)
        alpha = np.dot(x_train[t],v)
        h = 1/(1+np.exp(-alpha))
        pre = fwx(h,w)
        yty = np.dot(pre.reshape(1,yd),1-pre.reshape(yd,1))
        w -= a*yty*np.dot(h.reshape(hid,1),pre.reshape(1,yd)-y_train[t].reshape(1,yd))
        eph = -yty*np.dot(w,(pre.reshape(yd,1)-y_train[t].reshape(yd,1)))
        hth = np.dot(h.reshape(1,hid),1-h.reshape(hid,1))
        v += a*hth*np.dot(x_train[t].reshape(xd,1),eph.reshape(1,hid))
```

由于更新量是多个部分的乘积，所以任何一部分为零都可能导致这个情况。下面用打印的方法看看到底是哪一项为零：

```
while abs(l_new - l_old) >= threshold:
    l_old = l_new
    for i in range(k):
        t = random.randint(0, len(x_train)-1)
        alpha = np.dot(x_train[t],v)
        h = 1/(1+np.exp(-alpha))
        pre = fwx(h,w)
        yty = np.dot(pre.reshape(1,yd),1-pre.reshape(yd,1))
        w -= a*yty*np.dot(h.reshape(hid,1),pre.reshape(1,yd)-y_train[t].reshape(1,yd))
        eph = -yty*np.dot(w,(pre.reshape(yd,1)-y_train[t].reshape(yd,1)))
        hth = np.dot(h.reshape(1,hid),1-h.reshape(hid,1))
        v += a*hth*np.dot(x_train[t].reshape(xd,1),eph.reshape(1,hid))
        print("yty:",yty,"hth:",hth,"h:",h,"pre:",pre,"alpha",alpha)
        break
```

结果：

```
In [26]: runcell(33, 'D:/Machine_learning/sst2.py')
Have changed y into an array with shape (60000, 10)
Have changed y into an array with shape (60000, 10)
yty: [[3.08571923e-09]] hth: [[0.]] h: [1. 1. 1. 1. 1.
1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1. 1. 1. 1.
 1. 1.] pre: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.] alpha
[59.27388356 62.39035138 59.76773995 57.93931777
59.55241645 56.25503603
 62.05786863 56.33104998 58.96093862 60.44858364
54.51518684 63.64040672
 58.55875313 58.83156701 62.87113899 61.71710631
52.68518625 56.85695216
```

发现是$h^t(1-h)$这一项为零。而这是由于隐层每一维都是 1。为什么会都是 1 呢？h 的计算来自于 sigmoid 函数：

```
alpha = np.dot(x_train[t],v)
h = 1/(1+np.exp(-alpha))
```

如果 alpha 非常大的话，h 的取值就趋近于 1，超出精度范围就等于 1，无法更新系数了。刚刚也打印出来了 alpha 的值：

```
 1. 1.] pre: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.] alpha
[59.27388356 62.39035138 59.76773995 57.93931777
59.55241645 56.25503603
 62.05786863 56.33104998 58.96093862 60.44858364
54.51518684 63.64040672
 58.55875313 58.83156701 62.87113899 61.71710631
52.68518625 56.85695216
 61.28463844 58.33859473 55.16682424 60.17541851
56.78011341 55.92847087
 59.92532471 54.97473733 58.92056328 60.74766685
58.54730383 58.96525823
 55.59425345 56.63944678 50.06036914 57.98060893
58.43487563 55.58683051
 55.65881194 63.70770201 57.91854274 61.16518273
58.22091733 53.19251909
 57.48840384 60.72808902 58.47837585 58.39681641
64.47686264 58.49879192
```

发现 alpha 的值大概在 50-60，对于 sigmoid 函数来说，确实有些太大了。而 alpha 是 x 和 v 的线性组合。考虑到 x 是归一化到 0-1 之间的，而第一次计算使用的是随机产生的初始化 v：

```
v = np.random.random((x_train.shape[1],hid))
w = np.random.random((hid,y_train.shape[1]))
```

我初始化 v 是采用了 0-1 之间的随机数，同 785 维的 x 相乘，给出 50-60 的数几乎是必然的，所以是初始化的问题。将初始化的代码改为：

```
v = np.random.random((x_train.shape[1],hid)) - np.random.random((x_train.shape[1],hid))
w = np.random.random((hid,y_train.shape[1])) - np.random.random((hid,y_train.shape[1]))
```

这样初始的 v 和 w 都是-1 到 1 之间的数，与 x 相乘可以给出平均为 0 的值，不会再造成 sigmoid 函数=1 的情况。问题解决，神经网络模型正常运行。

所以为什么一开始没意识到是初始化的问题？因为我线性模型就是那样初始化的，而运行完全没问题。仔细对比两个模型更新系数的方式：

线性模型：

$$w_j = w_j + \alpha\,(\,y^{(i)} - f_x\big(x^{(i)}\big)\,)\,x_j^{(i)}$$

神经网络：

$$\delta_{\alpha_j} = h_j\,(\,1 - h_j\,)\sum_i \delta_{\beta_j}\,w_{ji}$$

$$v_{kj} = v_{kj} - \eta\,\delta_{\alpha_j}\,x_k$$

其实，用 0-1 初始化的系数处理线性模型时，第一次循环给出的也是基本全 1 的预测。但是由于线性系数的更新公式里没有$f(x)\,(\,1 - f(x)\,)$这样的项，所以不会出问题。神经网络则会受到这种初始化方式的影响。

六、感想与收获

作为生物系的学生，我上一次上和计算机相关的课还是大一的计算机科学导论。因为没有上过 C 语言，也没有任何其他编程语言的基础，所以为了做这一次的作业，就临时上网学习了据说最简单的 python。所以一周前我还完全不会编程，这个程序现在能跑出来，我已经超级有成就感了。

不过没有基础的话做起来真的累。学 python 的时候网上只会从最基本的开始教我，所以我一开始就只会用列表。后来上其他课的时候和一个学长聊起，他才让我学着用数组。我的程序已经很慢了，用列表的话估计就凉了。

所以这个过程真的好艰辛啊，我得沿着所有新手都犯过的错误全部走一遍。当别人在考虑算法的时候，我的程序还在因为缩进和括号报错。用到的所有函数都是上网现查的，有时候不熟悉的话会用错，真的好久都找不出来 bug。遇到内存不够之类的问题也完全不会处理，只能疯狂上网搜。

如果上过其他相关的计算机系基础课的话，大概就会知道怎样写效率高一点吧，我只会用最直觉的循环去做，自己都觉得自己的程序看起来非常繁冗，但是不知道该怎么办。以后应该还是要多学习一些这方面知识。

所以上这门课之前，我不仅是一个机器学习的小白，还完全是计算机和编程的小白。经过这一周赶作业的磨炼，我觉得我真的“成长”了。

虽然我的作业做的不是最好的，但我绝对是进步最大的。