第一课 -- Maker Club

前言

机器学习，就是让机器能够做到一些人类能做到的事情

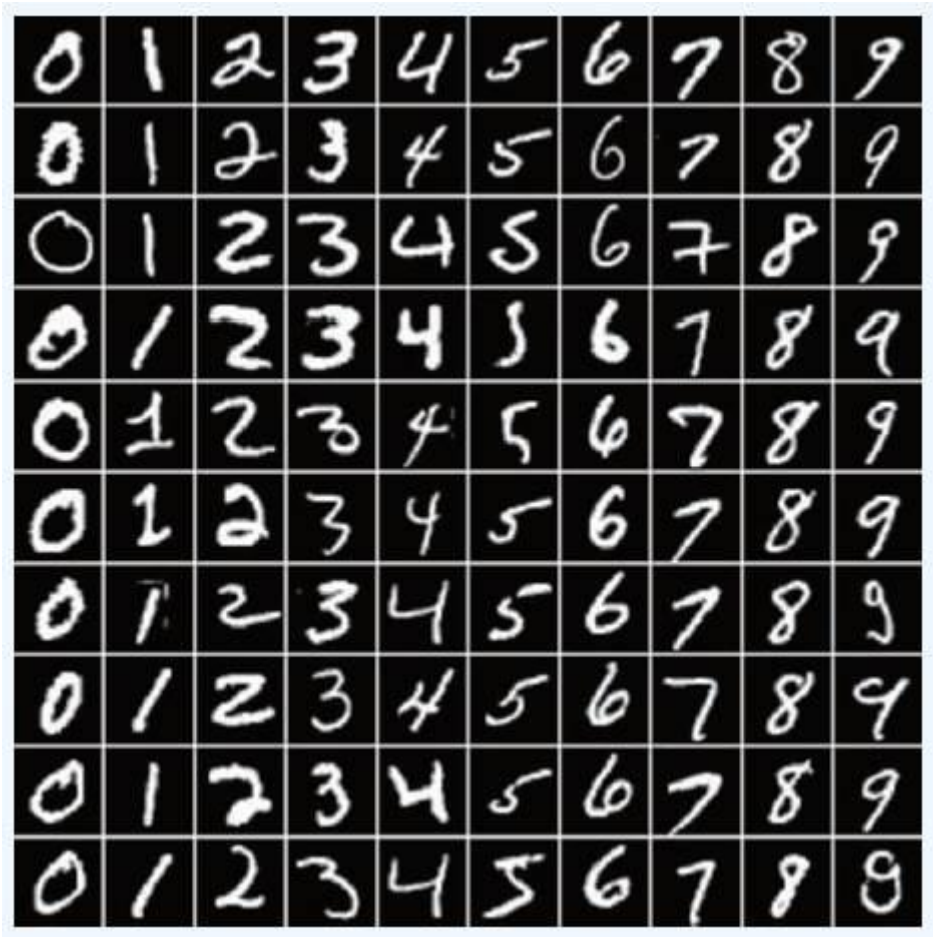
但是由于机器是精确的，死板的，一些人类做起来简单无比的事情机器却很难做好甚至很难做到。

比如图像分类，一个几岁的小孩就能很容易的分清路上的车，人，标志牌等等，而让机器做却很困难。

第一课我们会从一个最常用的图像分类入门任务开始，一步一步认知机器学习的一些原理。

首先介绍一下这个任务：

这个任务要用到的数据集为MNIST，是一个包含了60000多张手写数字的图片集，我们的任务则是**预测**一张图片上的数字到底是多少。



首先先明确的是，本任务的任务类型为**分类问题**。

知识点1：我们做机器学习，一般的任务类型有两种，分别是：

- 1.回归问题
- 2.分类问题

首先说说回归，回归问题是寻找一个模型使之能够学习到一组随机变量(自变量)与另一组随机变量(因变量)之间的关系，比如高中学习过的线性回归。回归一般能够预测任意的数值

其次是分类问题，分类问题其实也能算作是回归，但与回归最大的不同是：回归能够预测出任何数字，也就是说回归的预测是连续的。而分类问题，我们需要找到一个模型判断某个输入属于哪一个类别，此时，分类问题的输出为离散的，即指定输入属于哪个类别。

对于MNIST任务，我们希望能够有一个模型或算法能指出输入属于0~9之间的哪一个数字，即输出只有10种可能，因此为分类问题。

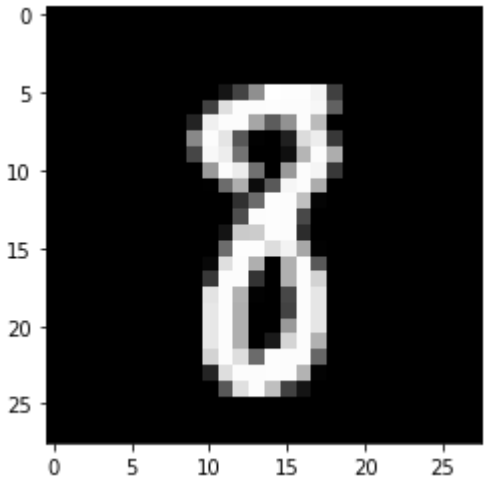
数据的处理

关于MNIST的数据读入，官方已经提供了一个文件：

In [1]:

```
import input_data
import matplotlib.pyplot as plt
import numpy as np
import math
%matplotlib inline

def show_img (np_arr):
    img = np.reshape(np_arr, (28, 28))
    plt.imshow (img, cmap='gray')
    plt.show()

mnist = input_data.read_data_sets("mnist_data/", one_hot=True)

show_img (mnist.train.images[50])
print (mnist.train.labels[50])
```

```
Extracting mnist_data/train-images-idx3-ubyte.gz
Extracting mnist_data/train-labels-idx1-ubyte.gz
Extracting mnist_data/t10k-images-idx3-ubyte.gz
Extracting mnist_data/t10k-labels-idx1-ubyte.gz
```




```
[0.  0.  0.  0.  0.  0.  0.  0.  1.  0.]
```

其中，每一张图片的大小都是固定的，为 (28 * 28), 储存成一个大小784的向量。

其次，每张图片对应的标签为一个One-Hot向量：解释一下

One-Hot：**独热码**, **直观来说就是有多少个状态就有多少比特，而且只有一个比特为1，其他全为0的一种码制。**

该例子中，因为图片的类别为8，因此One-Hot向量中只有$index = 8$的地方为1，其余都是0。

最初的尝试

我们可以简单想想，如何让计算机通过计算得到一张图片上的数字等于多少？

其中，一个最简单和最直接的想法就是，找相似度：如果计算机知道某张图片上的数字是'1'，那么是不是就可以把要预测的图片都拿来和这张图片进行对比，并计算出一个相似度，如果两张图片足够相似，则认为该图片和已知图片同类。

因此现在要解决的是，如何计算两张图片的相似度。

因为图片是以向量的方式保存的，因此我们能够据此给出一个很好的方法：

假设有两张图片，一张为已知图片P_t，一张为需要预测的图片P，则这两张图片对应的向量分别代表数据空间中的两个点，我们对这两个向量做差$P_t - P$，即可得到一个新的向量，一个由P指向P_t的向量，记作D，则P到P_t的欧式距离即为$|D| = \sqrt{D^T D}$，并且我们认为该距离可作为这两张图片相似度的衡量方法。

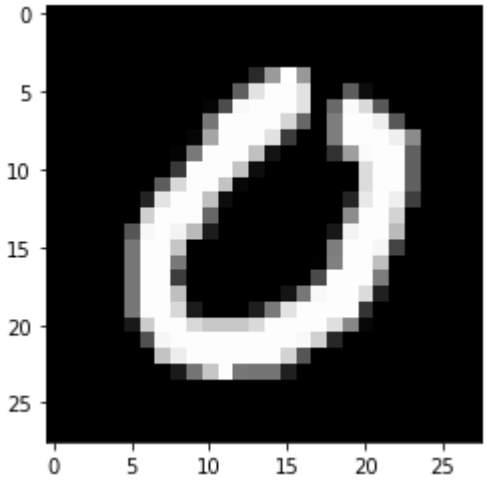
但是如果我们只有一张已知图片的话，可能并不能很好的代表全部的该类图片，比如有一张数字0，但是我们知道，数字0可以写的很圆，也可以写的瘦长一些：

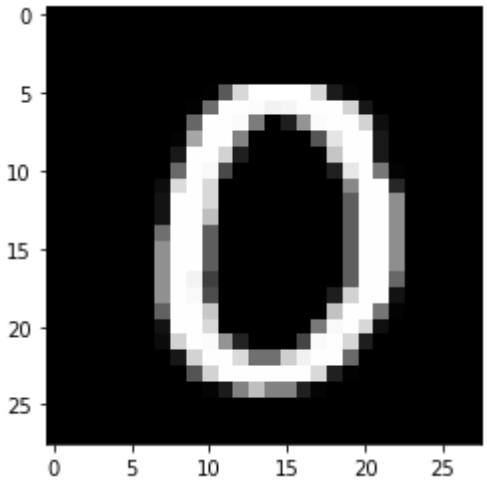
(下图中第一个0写的稍斜，而第二个比较正)。

因此，我们需要一个能代表某类别所有图片的向量！

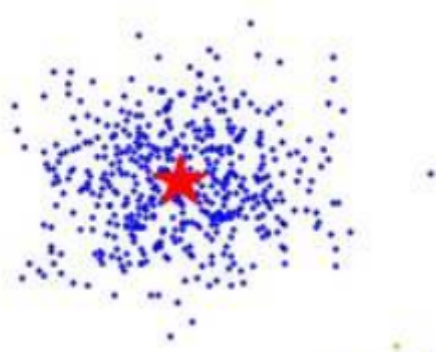
In [2]:

```
show_img (mnist.train.images[7])
show_img (mnist.train.images[10])
```





一个很简单却很有效的做法就是求出某类所有图片的平均值

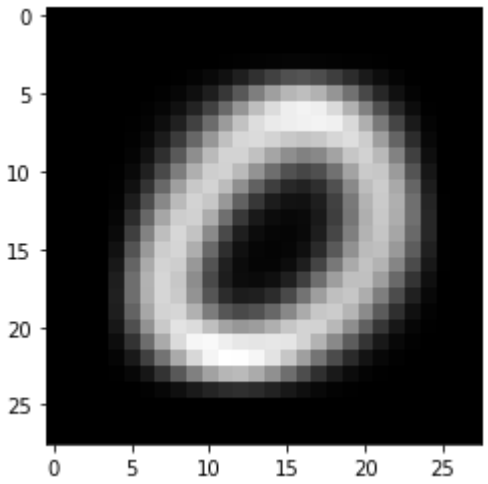


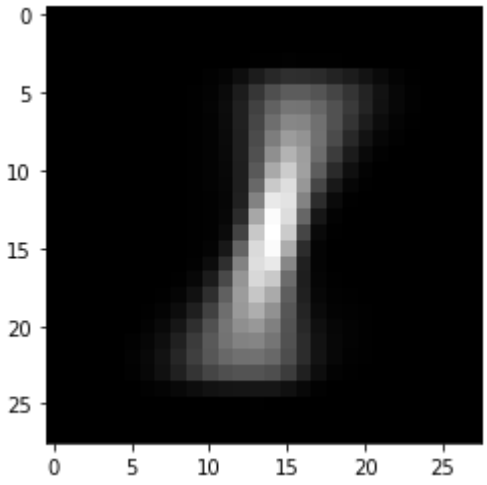
我们认为相同类的图片在样本空间中的分布会更加靠近，因此每一类的图片都会在样本空间中聚成不同的“团”，我们可以认为这个团的中心就是这个类别的平均代表，也就是该类中的所有样本到该点的距离的平均值最小。

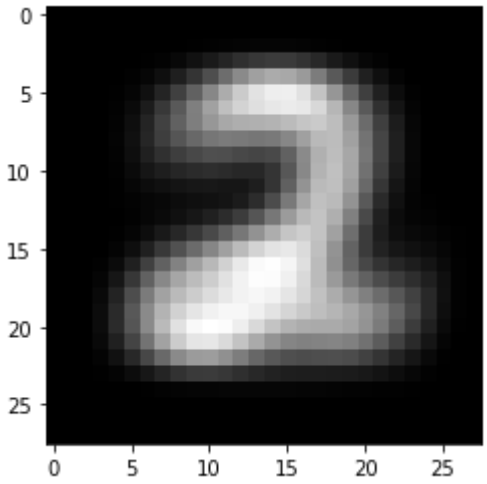
因此我们可以把需要预测的图片与该类别的中心点进行比较，而不是和单一的图片进行比较，这样我们就能考虑到某个类别的总体信息。

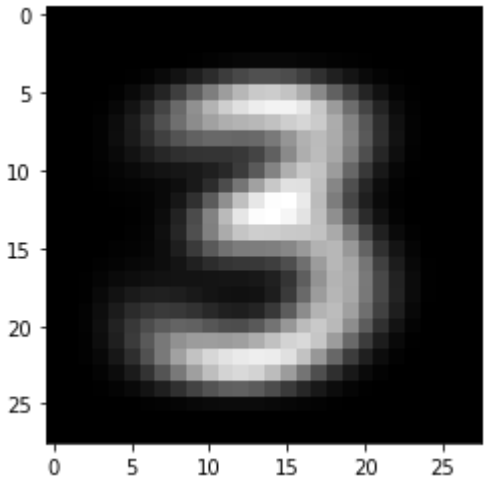
In [3]:

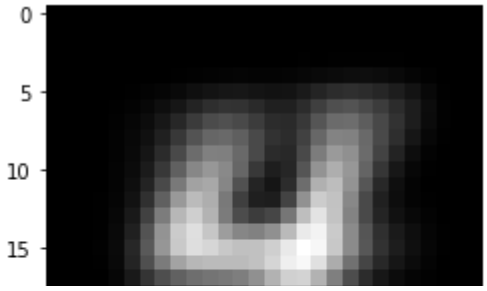
```
classes = [[] for i in range (10)]

train_image = mnist.train.images
train_label = mnist.train.labels

test_image = mnist.test.images
test_label = mnist.test.labels

for i in range(len (train_label)):
    classes[np.argmax (train_label[i])].append (train_image[i])

score = [0.0 for i in range (10)]

def get_feat_vec (index):
    vec_ = np.zeros (784)
    for x in classes[index]:
        vec_ += x
    vec_ /= len (classes[index])
    vec_ /= math.sqrt (np.dot (vec_, vec_)) # 化为单位向量
    return vec_

for i in range (10):
    feat_vect = get_feat_vec (i)
    show_img (feat_vect)
```

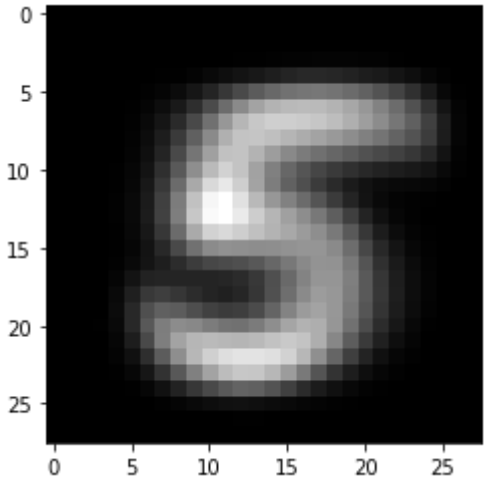


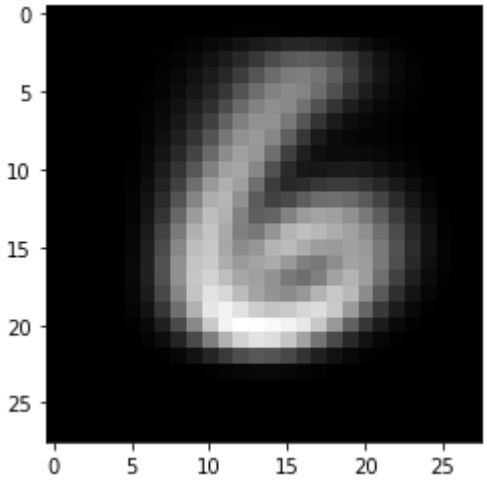

0
5
10
15
20
25
0
5
10
15
20
25

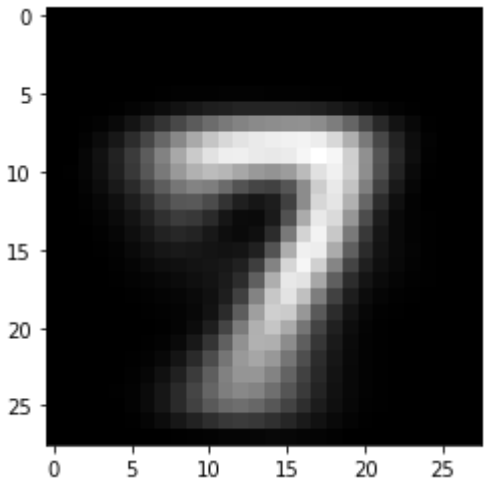

0
5
10
15
20
25
0
5
10
15
20
25

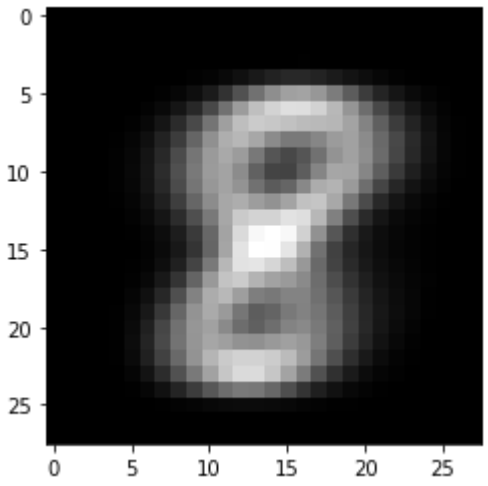

0
5
10
15
20
25
0
5
10
15
20
25

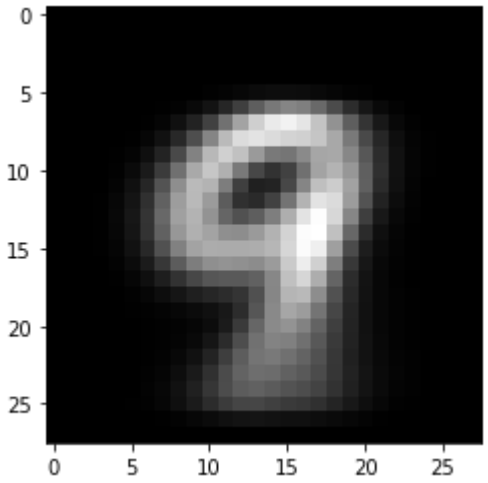

0
5
10
15


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25





这样，我们就能写出我们的第一个预测手写数字的程序：

- 1.首先得到每个类别的所有图片
- 2.将每个类别的图片的中心点求出来
- 3.将需要预测的图片与每个类别中心点相似度(即距离)算出来，并选出距离最小的一个，可认为预测图片与对应类为同类

In [4]:

```
feature = [get_feat_vec (i) for i in range (10)]
feature = np.asarray (feature)

acc = 0
for i in range(len (train_label)):
    true_label = np.argmax (train_label[i])
    x = train_image[i]
    x /= math.sqrt (np.dot (x, x))
    tmp = feature - x
    pred = [math.sqrt(np.dot (v, v)) for v in tmp]
    pred = np.argmin (pred)
    if true_label == pred:
        acc += 1
print ('训练集上准确率: %.2lf' % (acc / len (train_label)))


acc = 0
for i in range(len (test_label)):
    true_label = np.argmax (test_label[i])
    x = test_image[i]
    x /= math.sqrt (np.dot (x, x))
    tmp = feature - x
    pred = [math.sqrt(np.dot (v, v)) for v in tmp]
    pred = np.argmin (pred)
    if true_label == pred:
        acc += 1
print ('测试集上准确率: %.2lf' % (acc / len (test_label)))
```

```
训练集上准确率: 0.81
测试集上准确率: 0.82
```

可以看到，简单的想法实现的这个并不算难的程序，在这个任务上能达到80+的准确率，还算是很不错的。

让模型自己学习

虽然上面的算法已经有了一个看上去不错的结果，但是模型的判断方法过于主观，也就是我们先认为类型相同的数据之间“距离”会更近这个经验，而做机器学习，我们更希望算法能够自我学习如何分类数据，而尽量更少的从人那里获取先验。

先对上面所说的东西做一些小小的说明：

- 1.模型：模型其实就像函数一样，能够解决某些问题，但模型更多的是做预测，即能够通过某些从未见过的样本，预测出一个合理的输出
- 2.模型参数：参数是模型是否能预测正确的关键，在上面的任务中，我们可以看到模型是否能预测正确和feature参数有着很大的参数，因此让算法学习，其实就是让算法学习，找到一个较优的模型参数。
- 3.训练：模型学习和优化参数的过程叫做训练
- 4.监督学习和无监督学习：简单来说，监督学习就是说，我们能够获得用来训练模型的数据的真实标签，非监督学习则是无法获得训练数据的真实标签。例如MNIST数据集，我们对每张训练图片都能够获得对应的标签(即该图片代表的数值)。目前监督学习占据了机器学习算法的大部分。

可以看到，我们的手写数字识别应该算是一个监督学习，但是学习的步骤应该怎样呢？

回忆一下高中我们学习题型的过程，我们每当做出一道题后，都会去看看这道题的标准答案，来得知我们自己的过程是否正确。如果我们做错了，则需要加强对类似题型的理解和记忆。

监督学习也是如此，当模型预测出的标签和数据的真实标签不同时，算法会对模型进行一个惩罚，惩罚的力度视预测值与正确值的差异而定(即根据错误的度量而定)。

我们定义模型预测值为y，真实值为y_t

则对于回归问题，我们定义预测值与真实值的差距为：$Loss(y, y_t) = \frac{1}{2}||y_t - y||^2$，注意模型的输出一般是一个向量。可以看到该函数为一个凸函数，因此只有一最小值0，去最小值时有$y_t = y$，此时说明模型的预测和真实情况完全相同。且预测值与真实值差异越大，该函数值越大。(该函数叫做MSE，均方误差)

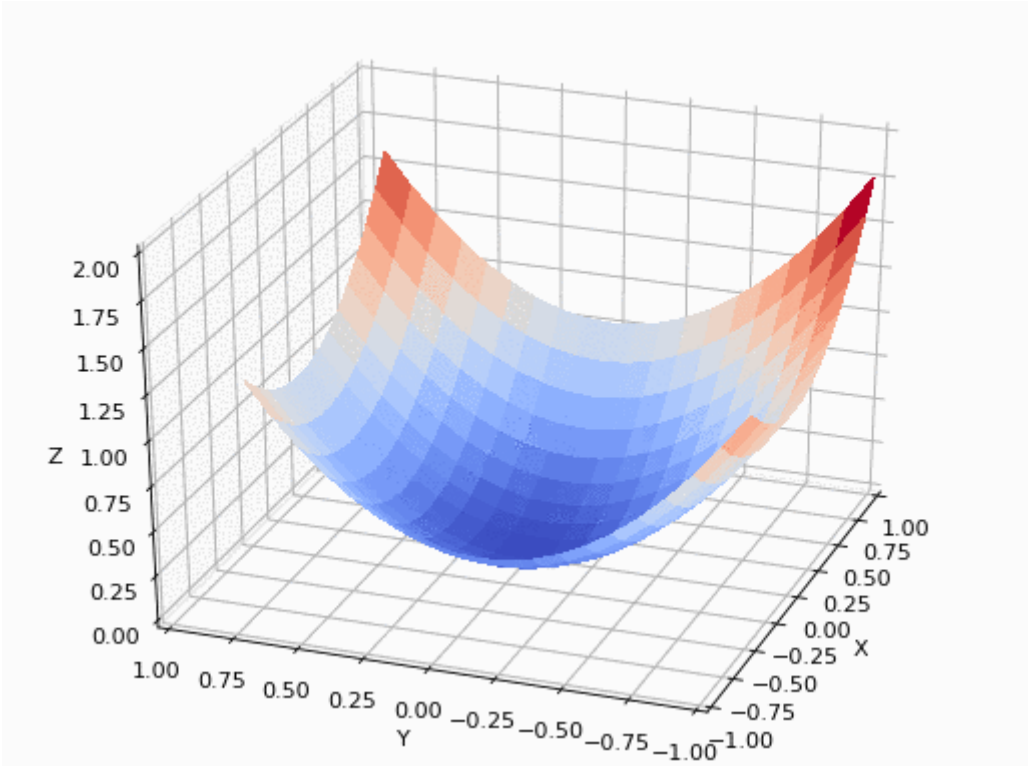
而对于分类问题，我们一般使用另一个函数来定义预测值与真实值的差距。由于在分类问题中，模型的输出一般是一个概率分布，即有$0 <= y_i <= 1$,且有$\sum y_i = 1$，定义：$Loss(y, y_t) = -\sum y_t log(y)$，该函数为信息论中的交叉熵函数，可用于衡量两个分布的差异，当$y = y_t$时，有函数值最小。(信息论之后会学到)

上面提到的Loss函数我们称为**损失函数**，也可以称为代价函数(cost function)，用来衡量模型预测时犯的错误的轻重。

虽然不同任务的Loss函数不一样，感觉很复杂，但是所有的损失函数都有一些共同点，就是当Loss函数越小时，我们模型犯的错误就越少，也就是模型越优秀。因此我们模型学习的思路就来了：找到合适的参数，去最小化Loss函数！

对于某些飞船非常简单的模型，我们可以推导公式，如利用拉格朗日乘子法等方法，直接找到取到Loss函数极值时的模型参数，但随着算法的越来越复杂，模型的参数也越来越多（就如我们上面的feature，是一个784*10的向量，也就是参数个数为7840个），让计算机直接解出最优参数显然没有那么现实。因此我们更多的选择了一种折中的办法：迭代法，也就是我们轮番的更新模型的参数，保证每次更新都能让loss下降一些。

其中一种方法叫做梯度下降法，这种方法很有意思，它和我们生活中的情形很相似。



比如我们在一个斜坡上放一个小球，根据常识我们知道小球一定是往坡下滚，梯度下降就是要模拟这种感觉，当前模型的参数在loss函数上就相当于一个小球，我们需要不断的往下移动来到达一个极小值点。要做到梯度下降，我们最需要解决的问题就是，当前应该往哪里移动？

根据微积分，导数的知识。对于一元函数，我们知道，函数某点的导数表示函数在该处变化率的大小，我们把导数想象为一个1维的向量，则导数为正对应了一个向右的向量，导数为负则对应了一个向左的向量。

根据演示，我们可以得知，当我们朝着当前导数向量的反方向移动时，函数值能够减少。

到了高维空间中，函数的所有变量的偏导数也构成了一个向量：梯度。梯度的方向为函数值增长最快的方向，梯度的模长则为该放向上函数的变化率。和1维一样，我们只需要沿着梯度的反方向进行移动，就能保证函数值越来越小。虽然参数量大时梯度的求解也需要不少算力，但是相比直接求解还是容易了许多。

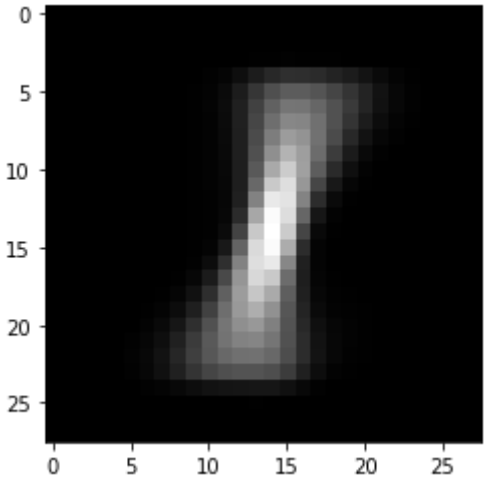
需要用到的知识点：普通函数的求导公式，复合函数的求导规则(链式法则)

所以可以大致得到一个算法步骤：

- 1.选择一个模型并对模型的参数进行初始化，参数的初始化也对模型有着很大的影响，但是这部分以后再说。
- 2.选择一个合适的损失函数
- 3.将一个样本投入模型，得到预测以及模型对该样本的loss值
- 4.求出模型中每个参数的梯度
- 5.最后对每个参数向着梯度的反方向更新一小步。
- 如果loss值降到一个可接受的范围或训练次数达到某个值后返回，否则返回步骤3

注：步骤3一般称为前向传播，步骤5一般称为反向传播

首先进行模型的选择，我们选择一个size为$(784, 10)$的矩阵，可以看成由10个特征向量构成。我们将输入样本与权重矩阵进行矩阵乘法操作，可以看作样本在和每一个特征向量做点积操作(点积可以判断两个向量是否相关，或者从图像方面解释，特征向量中的每一个分量表示的是样本图片中对应的像素的重要程度)



矩阵乘法后得到的是一个大小为10的向量，分别代表样本是哪个类别的置信度。

此时输出的向量可能有正有负，有大有小。而因为这是一个分类问题，因此我们想把输出向量转成一个概率分布，这里需要用到softmax函数，简单介绍下：

softmax可以把一个向量转成对应的概率分布，满足：向量中数值大的分量转化出来的结果，对应的概率也会更大。 softmax的表达式为：

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

看上去挺复杂的，但是其实这个可以看作另一个东西的稍微改进版：

$$P_i = \frac{x_i}{\sum_j x_j}$$

对于多次采样，我们可以认为x的概率为x发生的次数除以所有事情发生的次数，而对于置信度，我们也可以类似的进行，但是经过矩阵乘法后的向量有正有负，上式对于负数就不能完美的处理了(比如求和中最终分母为0)，而softmax加入了e^x(能使用e^x是因为e^x是一个单调递增函数)，消除了负数的影响，并且经过Softmax处理后的数据，能达到置信度高的项和置信度低的项之间的概率差距更大,相当于变相的加大了惩罚力度，让模型训练的更快。

关于Softmax函数之后会有更详细更全面的讲解

接下来就可以选择损失函数了，这里就使用之前说的分类专用的交叉熵损失函数。

选择完损失函数，接下来就是进行前向传播了：

In [5]:

```
input_num = 784 # 28 * 28 * 1 的图片
output_num = 10 # 输出为10个类别的评分

weight = np.random.rand (input_num, output_num) # 权重

def softmax (pred_arr):
    # 将一个预测的得分转换为一个概率分布
    max_n = max (pred_arr)

    for x in range (len (pred_arr)):
        pred_arr[x] -= max_n # 这是一个小优化，防止指数操作爆炸
    pred_arr = np.exp (pred_arr)
    sum_n = sum (pred_arr)

    ret = []
    for x in range (len (pred_arr) - 1):
        ret.append (pred_arr[x] / sum_n)
    ret.append (1.0 - sum(ret))
    return ret

def loss_func (pred, label): # 损失函数：交叉熵
    return - np.dot (label, np.log2 (pred))

def forward (data): # 前向传播
    ret = np.matmul (data, weight)
    return ret
```

接下来就是求导了：求导较为复杂，都是我手动推到过一遍的。有闲心的话可以试试自己推理一遍。 如果不想自己推导的话，则使用现成的结果就好，但是尽量需要自己能推导一遍！因为以后使用和学习不同的模型可能需要自己计算一些公式。

$$L = -\sum_i y_{ti}\ log\ y_i$$

则有

$$\frac{\partial L}{\partial y_i} = -\frac{y_{ti}}{y_i}$$

记

$$Softmax(z_j) = S_j = \frac{e^{z_j}}{\sum_i^n e^{z_i}}$$

$$Softmax(z_j) = \frac{e^{z_j}}{\sum_i^n e^{z_i}}$$

$$\frac{\partial S_j}{\partial z_i} = \begin{cases} -S_i \cdot S_j & \text{if } i \neq j \\ S_j(1 - S_j) & \text{if } i = j \end{cases}$$

$$\frac{\partial L}{\partial z_i} = \sum_j \frac{\partial L}{\partial S_j} \frac{\partial S_j}{\partial z_i} = \sum_{j \neq i} (\frac{\partial L}{\partial S_j} \frac{\partial S_j}{\partial z_i}) + \sum_{j=i} (\frac{\partial L}{\partial S_j} \frac{\partial S_j}{\partial z_i})$$

$$\begin{cases} -\frac{y_{tj}}{S_j} \cdot -S_i S_j = y_{tj} \cdot S_i & \text{if } i \neq j \\ -\frac{y_{tj}}{S_j} \cdot S_j(1 - S_j) = -y_{tj}(1 - S_j) & \text{if } i = j \end{cases}$$

$$\sum_{i \neq j} (y_{tj} S_i) - y_{ti}(1 - S_i) = \sum_{i \neq j} S_i y_{tj} + S_i y_{ti} - y_{ti} = S_i - y_{ti}$$

$$or = \begin{cases} S_i - 1 & \text{if } i = pos \\ S_i & \text{if } i \neq pos \end{cases}$$

$$Z = W \vec{x} + \vec{b}$$

$$\frac{\partial Z_i}{\partial b_i} = 1$$

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial Z_i} \frac{\partial Z_i}{\partial b_i} = \frac{\partial L}{\partial Z_i} = d\_S_i$$

$$Z_i = \sum_j x_j W_{j,i}$$

$$\frac{\partial Z_i}{\partial W_{j,i}} = x_j$$

$$\frac{\partial L}{\partial W_{j,i}} = d\_S_i x_j = x^T d\_S$$

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial Z_j} \frac{\partial Z_j}{\partial x_i}$$

$$\frac{\partial Z_j}{\partial x_i} = W_{i,j}$$

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial Z_j} \frac{\partial Z_j}{\partial x_i} = \sum_j d\_S_j \cdot W_{i,j} = d\_SW^T$$

根据上面的公式，我们就可以求出参数对应的梯度：

In [6]:

```
def backword (pred, label, invec):
    global weight
    global bias
    pos = -1
    for x in range(len(label)):
        if label[x] == 1:
            pos = x
            break
    d_L = - label[pos] / pred[pos] # Loss函数对y的梯度
    d_S = np.zeros_like (pred) # Softmax对z的梯度
    for i in range(len(pred)):
        if i == pos:
            d_S[i] = pred[i] - 1
        else:
            d_S[i] = pred[i]



    invec = np.reshape (invec, (1, 784))
    d_S = np.reshape (d_S, (1, len(pred)))
    d_w = np.matmul (invec.T, d_S) # 参数的梯度

    weight -= d_w * learning_rate # 这里乘以一个较小的数值，是为了防止移动过快而跨过了最小值点
    # 也就是学习过程需要稳一些。
```

In [7]:

```
# 接下来就是训练，这里定义训练四十万轮
learning_rate = 0.005
for i in range (400000):
    data = mnist.train.next_batch (1)
    pred = softmax(forward (data[0][0]))
    if i % 10000 == 0:
        print ('训练第%06d, loss:%.6f' % (i, loss_func (pred, data[1][0])))
    backword(pred, data[1][0], data[0][0])

test_img, test_label = mnist.test.images, mnist.test.labels

tot = len (test_label)
acc = 0
for i in range (tot):
    pred = softmax(forward (test_img[i]))
    if np.argmax (pred) == np.argmax (test_label[i]):
        acc += 1

print ('准确率: %.2f' % (acc / tot))
```

```
训练第000000, loss:3.193604
训练第010000, loss:3.366739
训练第020000, loss:3.439745
训练第030000, loss:0.803797
训练第040000, loss:1.259554
训练第050000, loss:0.687568
训练第060000, loss:1.397768
训练第070000, loss:1.833548
训练第080000, loss:0.975134
训练第090000, loss:0.469604
训练第100000, loss:0.484063
训练第110000, loss:1.674823
训练第120000, loss:0.745722
训练第130000, loss:1.281666
训练第140000, loss:0.392521
训练第150000, loss:0.658302
训练第160000, loss:0.778721
训练第170000, loss:1.942692
训练第180000, loss:1.386795
训练第190000, loss:2.119086
训练第200000, loss:1.094315
训练第210000, loss:0.773151
训练第220000, loss:0.792211
训练第230000, loss:0.325613
训练第240000, loss:0.102066
训练第250000, loss:0.454376
训练第260000, loss:2.507249
训练第270000, loss:0.136624
训练第280000, loss:0.111959
训练第290000, loss:1.274804
训练第300000, loss:1.783338
训练第310000, loss:0.184450
训练第320000, loss:0.158458
训练第330000, loss:0.538585
训练第340000, loss:1.878295
训练第350000, loss:0.038215
训练第360000, loss:0.580877
训练第370000, loss:0.944173
```

```
训练第380000, loss:0.079440
训练第390000, loss:0.379781
准确率: 0.89
```

可以看到准确率达到了89%，最高能到达92%，这个模型仅为一个很简单的模型，训练方法也不是最好，但是也达到了很好的效果。当然，这个“简单”的模型代码量也不少了，不过之后会提到一些机器学习库，能够自动微分，就省去了我们手动推导梯度的过程了！能大大的减少代码量(如本模型使用tensorflow高阶API仅需7行即可完成训练)，而之后我们会学习到更多的方法，更多的模型，更多的算法。去解决更多的实际问题。