# Machine Learning Assignment 1



# 定义SVM

1. 编程环境：python2.7+opencv3.20+win10
   为了方便的利用图像处理工具，我选择的是opencv3.20。SVM模型也是opencv中的。为了方便之后的调用我将SVM模型和相应的训练和测试函数都封装成了一个自定义的SVM类，定义如下：

```
# -*- coding:utf-8 -*-
import  cv2
import struct
import numpy as np
import time
class svm():
	def __init__(self,c=10.0,gamma=0.01,kernel=1):

		# kernel
		# kernel=1   LINEAR
		# nernel=2   RBF
		# kernal=3

		self.model=cv2.ml.SVM_create()
		self.model.setType(cv2.ml.SVM_C_SVC)   # SVM类型
		if kernel==1:
			self.model.setKernel(cv2.ml.SVM_LINEAR)
			print 'Linear model is built'
		else:
			self.model.setKernel(cv2.ml.SVM_RBF)
			print 'Rbf model is built'
		self.model.setC(c)
		self.model.setGamma(gamma)

		print 'New svm model has been built'

	def train(self,train_data,train_label):
		# train the model and return train time
		t1=time.time()
		print 'The Svm is training ... \n'
		self.model.train(train_data,cv2.ml.ROW_SAMPLE,train_l
abel)
		t2=time.time()
		print 'Cost time is %s s \n' % (t2-t1)
		return (t2-t1)

	def save(self,name):
		path='D:/OneDrive - sjtu.edu.cn/Desktop/number_svm/'
		full_name=path+name
		self.model.save(full_name)

	def load(self,name):
		path='D:/OneDrive - sjtu.edu.cn/Desktop/number_svm/'
		full_name=path+name
		self.model=cv2.ml.SVM_load(full_name)

	def predict(self, test_data, test_label):
		[res,result]=self.model.predict(test_data)
```

```
                accuracy=result==test_label
                acc=sum(accuracy)
                acc=float(acc)/(np.size(test_label,0))
                print 'accuracy is %s' % acc
                return result, acc, accuracy
```

为了方便之后对模型只进行测试，在svm类定义中添加 Test 函数这样之后可以直接调用函数对训练的模型进行测试。

2. 加载数据
   因为 MNIST 提供的是二进制格式，我按照官网上的介绍对数据进行加载

**TRAINING SET LABEL FILE (train-labels-idx1-ubyte):**

| [offset] | [type] | [value] | [description] |
|---|---|---|---|
| 0000 | 32 bit integer | 0x00000801(2049) | magic number (MSB first) |
| 0004 | 32 bit integer | 60000 | number of items |
| 0008 | unsigned byte | ?? | label |
| 0009 | unsigned byte | ?? | label |
| ..... | | | |
| xxxx | unsigned byte | ?? | label |

The labels values are 0 to 9.

**TRAINING SET IMAGE FILE (train-images-idx3-ubyte):**

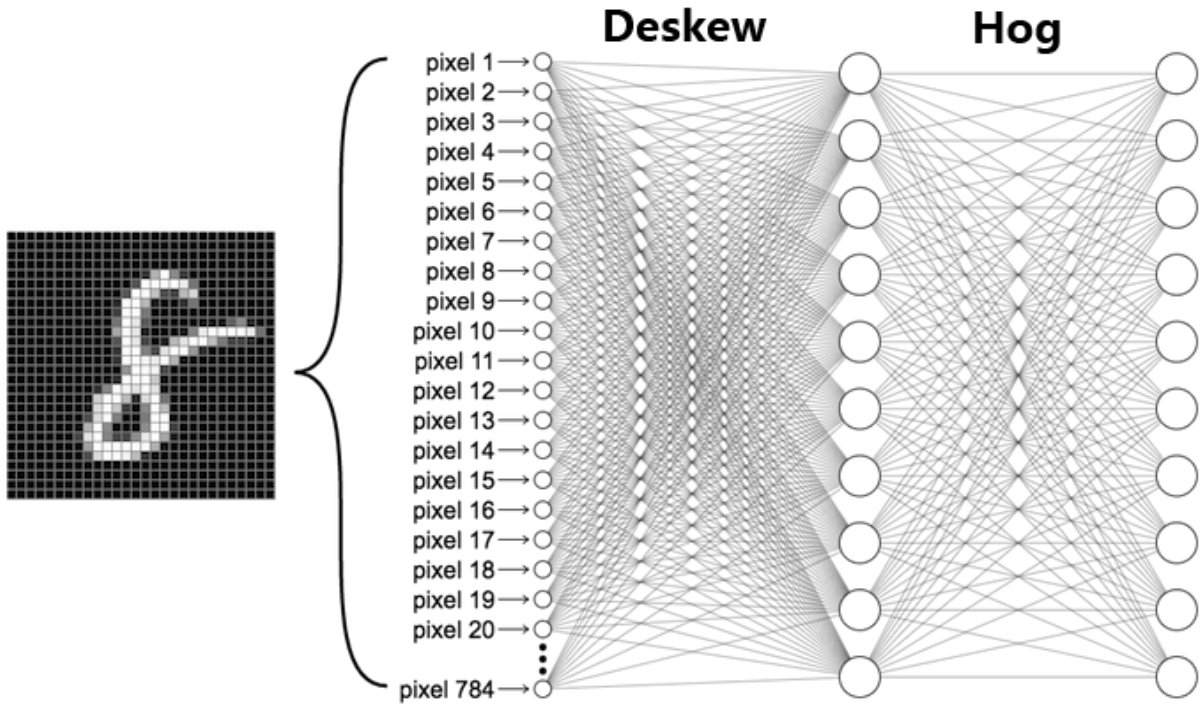
| [offset] | [type] | [value] | [description] |
|---|---|---|---|
| 0000 | 32 bit integer | 0x00000803(2051) | magic number |
| 0004 | 32 bit integer | 60000 | number of images |
| 0008 | 32 bit integer | 28 | number of rows |
| 0012 | 32 bit integer | 28 | number of columns |
| 0016 | unsigned byte | ?? | pixel |
| 0017 | unsigned byte | ?? | pixel |
| ..... | | | |
| xxxx | unsigned byte | ?? | pixel |

Pixels are organized row-wise. Pixel values are 0 to 255. 0 means background (white), 255 means foreground (black).

Mnist数据格式

```
def load_image(filename):
    print "load image set"
    binfile = open(filename, 'rb')
    buffers = binfile.read()

    head = struct.unpack_from('>IIII' , buffers ,0)
    print "head,",head

    offset=struct.calcsize('>IIII')
    imgNum=head[1]
    width=head[2]
    height=head[3]
    #[60000]*28*28
    bits=imgNum*width*height
    bitsString='>'+str(bits)+'B' #like '>47040000B'

    imgs=struct.unpack_from(bitsString,buffers,offset)

    binfile.close()
    imgs=np.reshape(imgs,[imgNum,width*height])
    print "load imgs finished"
    return np.float32(imgs/255.0)

def load_label(filename):
    print "load label set"
    binfile=None
    binfile=  open(filename, 'rb')
    buffers = binfile.read()

    head = struct.unpack_from('>II' , buffers ,0)
    print "head,",head
    imgNum=head[1]

    offset = struct.calcsize('>II')
    numString='>'+str(imgNum)+"B"
    labels= struct.unpack_from(numString , buffers , offset)
    binfile.close()
    labels=np.reshape(labels,[imgNum,1])

    #print labels
    print 'load label finished'
    return labels
```

Load_image 和 load_label两个函数可以分别用来加载数据和数据对应的标签。原始图像为尺寸$28 \times 28$，值在0~255之间的灰度图像,为了方便数据的表示，加载过程中将矩阵转化为向量，同时对数值进行归一化处理。




](./mnist_1layer.png)

# Linear kernel Model

在预处理得到理想的数据之后，分别构造了线性核和高斯核两种SVM模型进行训练测试。在线性模型中模型的核函数为：

$$\kappa(x_i, x_j) = x_i^T x_j$$

要进行优化的目标函数为：

$$min_{wb} \frac{1}{2} \|\omega\|^2 + C\Sigma(i=1)^m l_{01}(y_i(\omega^T x_i + b) - 1)$$

# 参数优化

在试运行之后发现规模较小的时候（训练样本为1000）模型的运行速度较短，但是在使用默认参数情况下识别的准确率不高。为了优化参数，需要对参数进行优化，找到最佳参数，这里直接使用格点搜索的的方法进行优化，而且在线性模型中主要的参数只有C，在优化线性模型的时候主要是对参数c进行优化。

```
def grid_search(c,gamma,train_data,train_label,test_data,test_label):
   s1=np.size(c,0)
   s2=np.size(gamma,0)
   acc=np.zeros((s1,s2))

   for i in range(s1):
        for k in range(s2):
                    print 'c=%s, gamma=%s'% (c[i],gamma[k])

                    model=svm(c[i],gamma[k],2)
                    model.train(train_data,train_label)
                    result, acc[i][k], accuracy= model.predict(test_data,test_label)
                        print '%s%% finshed' %  ((i*s2+k+1)/(s1*s2/100.0))

            # name='gamma=%s_c=%s.xml' % (c(k),gamma(i))
   best_para=np.amax(acc)
   for p in range(50):
    for q in range(50):
        if acc[p][q]==best_para:
                            break
   best_c=c[p]
   best_g=gamma[q]
   np.save('acc_search.npy',acc)
   return best_c, best_g
```

# 模型分析

为了得到在不同训练样本下模型的表现，对训练集做了新的划分，得到样本数分别是10000， 20000， 30000， 40000， 50000， 60000的训练集。

## 训练时间和准确率

之后按照之前预处理得到的最优化的参数产生六个新的模型，之后对这些模型进行进行训练测试，可以得到如图 Fig1 所示的训练时间以及准确率随着样本数量的变化。

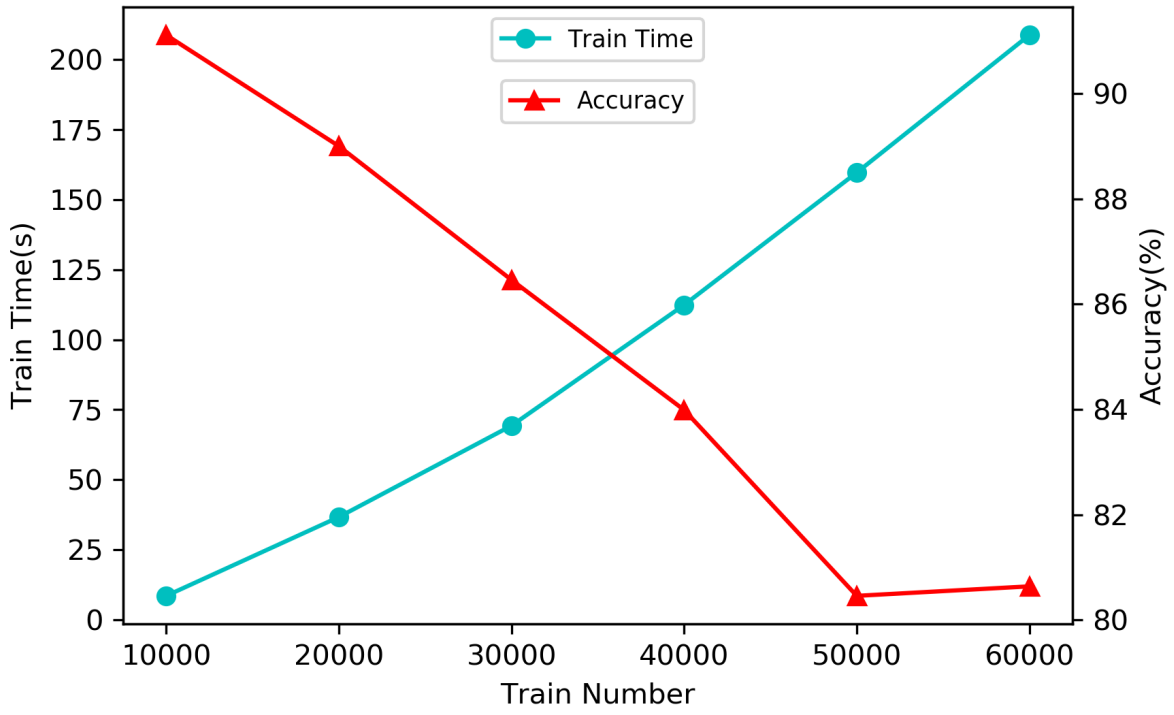



Fig1 Trian Time and Accuracy

根据Fig1 看出

1. 随着训练样本的增加，训练时间线性的增加，二者之间的关系如下：

$$y = 0.031x$$

2. 到一个比较奇怪的现象，在Linear kernel 中随着训练样本数量的增加，训练的准确率却在不但下降，知道样本数量达到50000的时候才有再次上升的趋势。

> 一方面是因为在优化参数的时候样本的选取是随机的，另外一方面在优化参数的时候训练集比较小，这样就导致优化出的参数可能对全局不适用。

## 对不同数字识别的准确率

为了能够更详细的分析模型的性能，我们计算模型识别每一类数字的准确率，可以得到如Fig 2的图像

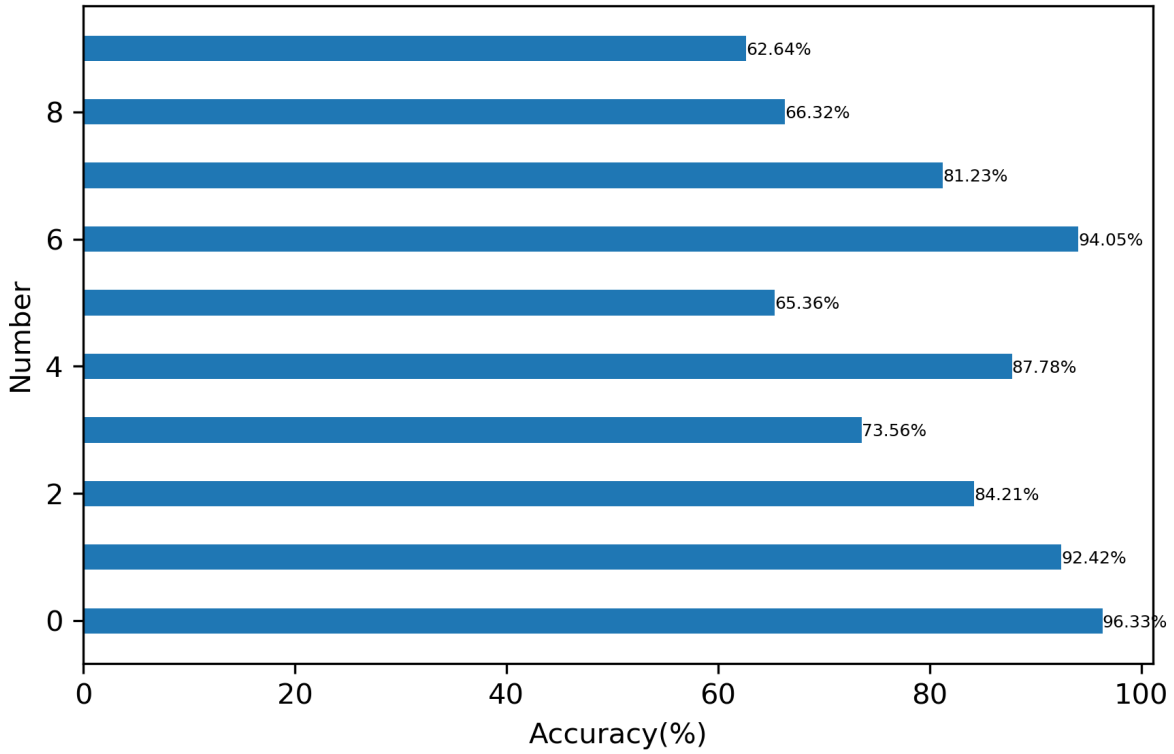



Fig2 不同数字识别的准确率

根据Fig5 可以看出 Linear Kernel 模型对有些数字如 5，8, 9 识别的准确率很低，只能到到60%多，但是对于其他数字识别的准确都比较理想。
为了进一步研究这些数字识别准确率低的原因，我们对识别结果做进一步分析，可以得到如下的识别错误矩阵如图 Fig3，矩阵（i, j）的元素代表在训练样本中将数字 i 识别为 j 的次数。

| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Total Error |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 7 | 1 | 0 | 7 | 8 | 3 | 1 | 0 | 27 |
| 1 | 0 | 0 | 2 | 1 | 0 | 2 | 3 | 1 | 6 | 0 | 15 |
| 2 | 8 | 11 | 0 | 13 | 11 | 4 | 12 | 11 | 35 | 3 | 108 |
| 3 | 3 | 2 | 29 | 0 | 2 | 27 | 2 | 9 | 25 | 7 | 106 |
| 4 | 2 | 1 | 11 | 0 | 0 | 0 | 7 | 5 | 2 | 22 | 50 |
| 5 | 12 | 5 | 7 | 61 | 9 | 0 | 13 | 2 | 21 | 6 | 136 |
| 6 | 12 | 3 | 11 | 1 | 10 | 13 | 0 | 0 | 2 | 1 | 53 |
| 7 | 2 | 8 | 25 | 17 | 10 | 0 | 0 | 0 | 3 | 32 | 97 |
| 8 | 7 | 21 | 9 | 39 | 10 | 35 | 10 | 6 | 0 | 10 | 147 |
| 9 | 8 | 7 | 1 | 16 | 52 | 4 | 1 | 55 | 6 | 0 | 150 |

Fig3 识别误差矩阵

根据上面的矩阵我们可以看到数字 5 经常被误识别为 3，次数高达61次， 而数字 9 则经常被识别为 4 （52次）和7（55次）。

> 一方面是因为线性模型的处理能力有限，还有客观因素，因为这些数字在外形上有一定的相似性，而且训练集中部分数字很相似，因此很难进行区分，我们会在之后的RBF模型中对这一现象做进一步讨论。

# RBF kernel

我们选择的主力方法是 RBF kernel构成的SVM模型，和之前的线性模型相比它的效果更好。RBF也就是高斯核对应的和函数为：

$$\kappa(x_i, x_j) = exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2})$$

优化的目标函数为：

$$max\Sigma_{i=1}^{m}\alpha_i - \frac{1}{2}\Sigma_{i=1}^{m}\Sigma_{j=1}^{m}\alpha_i\alpha_j y_i y_j \kappa(x_i, x_j)$$

# 参数优化

和之前线性模型的优化方法不同， 使用 RBF kernel 时候主要的参数有两个 c 和 $\gamma$，我们采用格点搜素的方法寻参。

```
def grid_search(c,gamma,train_data,train_label,test_data,test_label):
   s1=np.size(c,0)
   s2=np.size(gamma,0)
   acc=np.zeros((s1,s2))

   for i in range(s1):
        for k in range(s2):
                    print 'c=%s, gamma=%s'% (c[i],gamma[k])

                    model=svm(c[i],gamma[k],2)
                    model.train(train_data,train_label)
                    result, acc[i][k], accuracy= model.predict(test_data,test_label)
                        print '%s%% finshed' %  ((i*s2+k+1)/(s1*s2/100.0))

            # name='gamma=%s_c=%s.xml' % (c(k),gamma(i))
   best_para=np.amax(acc)
   for p in range(50):
    for q in range(50):
        if acc[p][q]==best_para:
                            break
   best_c=c[p]
   best_g=gamma[q]
   np.save('acc_search.npy',acc)
   return best_c, best_g
```

在实际寻找参数的过程中，选择的搜索格点范围如下：

```
gamma=np.linspace(0.01,0.050,50)
c = np.linspace(15,45,50)
best_c,best_g, acc_search=grid_search(c, gamma, train_data[:10000][:], train_labels[:10000], test_datas, test_labels)
```

在搜索的过程中，选择训练样本数量为1000，利用上面搜索得到的最优化参数为 c=21, $\gamma = 0.031$，最大准确率为$best_accuracy = 0.938$。为了观察准确率和参数之间的关系，可以绘制准确率和构成的热力图 Fig4

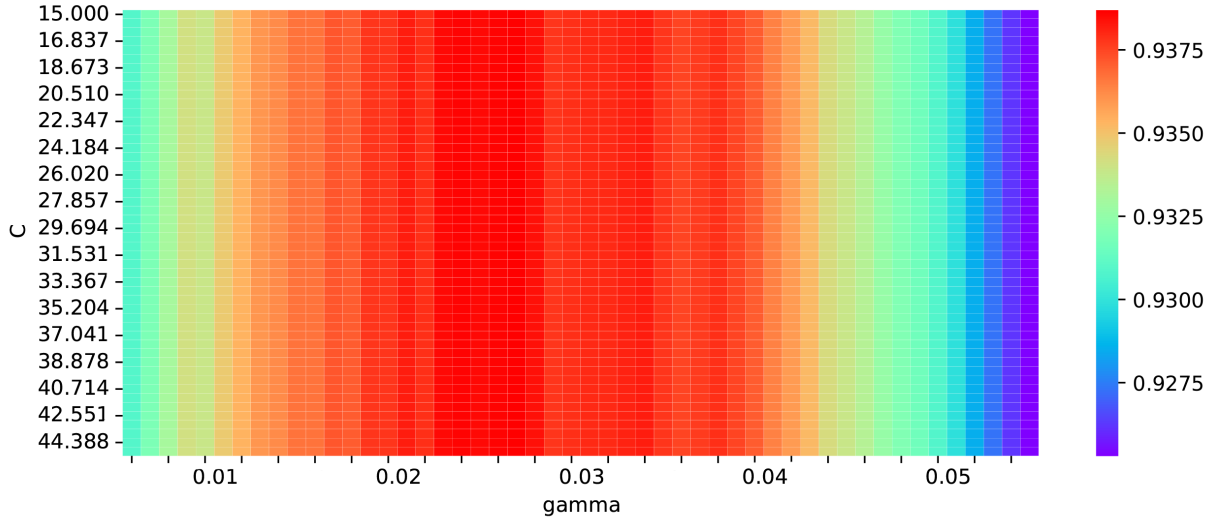



Fig4 寻参热力图

根据Fig4，可以发现，采用 RBF kernel的时候，模型对参数$\gamma$十分敏感，对参数c的敏感程度不高，因为

> 参数敏感程度分析

# 模型分析

## 训练时间和准确率

和之前一样，这次模型训练也是分成六组进行，进过训练识别准确率最高可以达到$98.58\%$，训练耗时和模型准确率随着时间的变化趋势如图Fig5

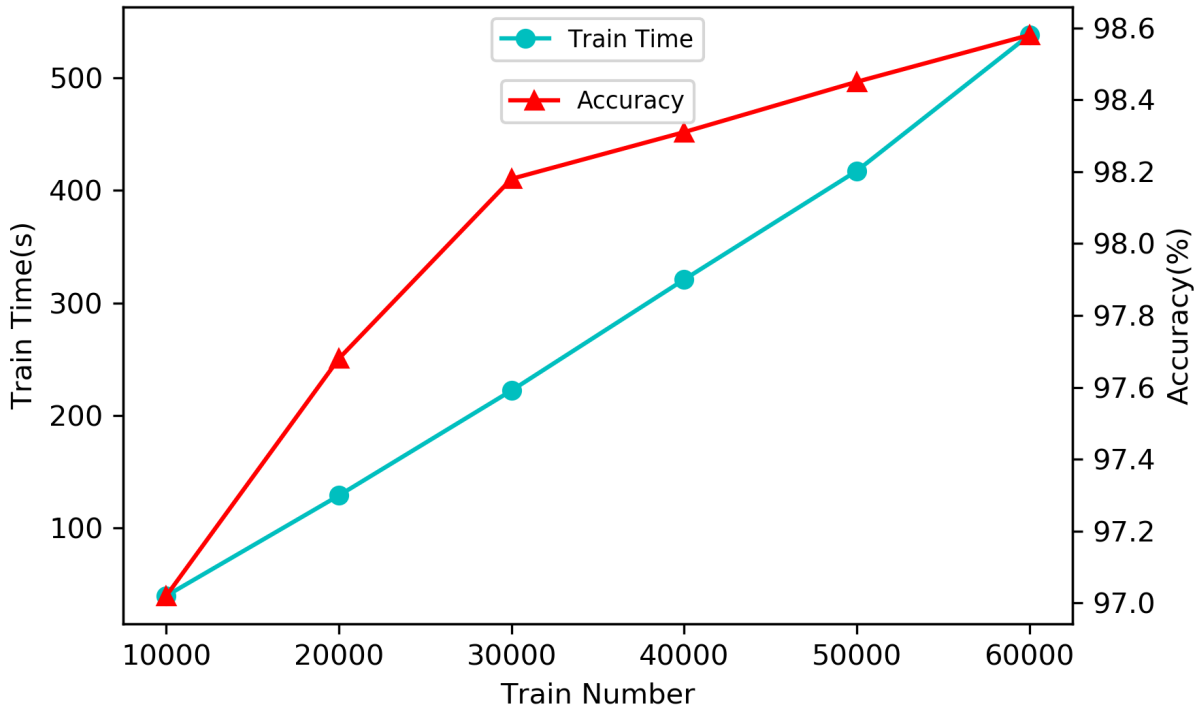



Fig 5 RBF kernel 训练时间和准确率

根据上面的图像可以看出：

1. 采用 RBF kernel 的时候模型的分类效果很好，即使在小样本数量（10000）的情况下，模型的准确率也可以达到$97.02\%$，随着训练样本的增加模型的训练准确率也随之继续升高,最高的训练准确率能够达到 $98.58\%$.
2. 训练时间和样本数目之前的关系基本是线性的，我们可以对只进行模拟得到二者之间的关系r如下：

$$y = 0.081 * x$$

## 对不同数字识别的准确率

为了能够更详细的分析模型的性能，我们计算模型识别每一类数字的准确率，可以得到如Fig6 的图像

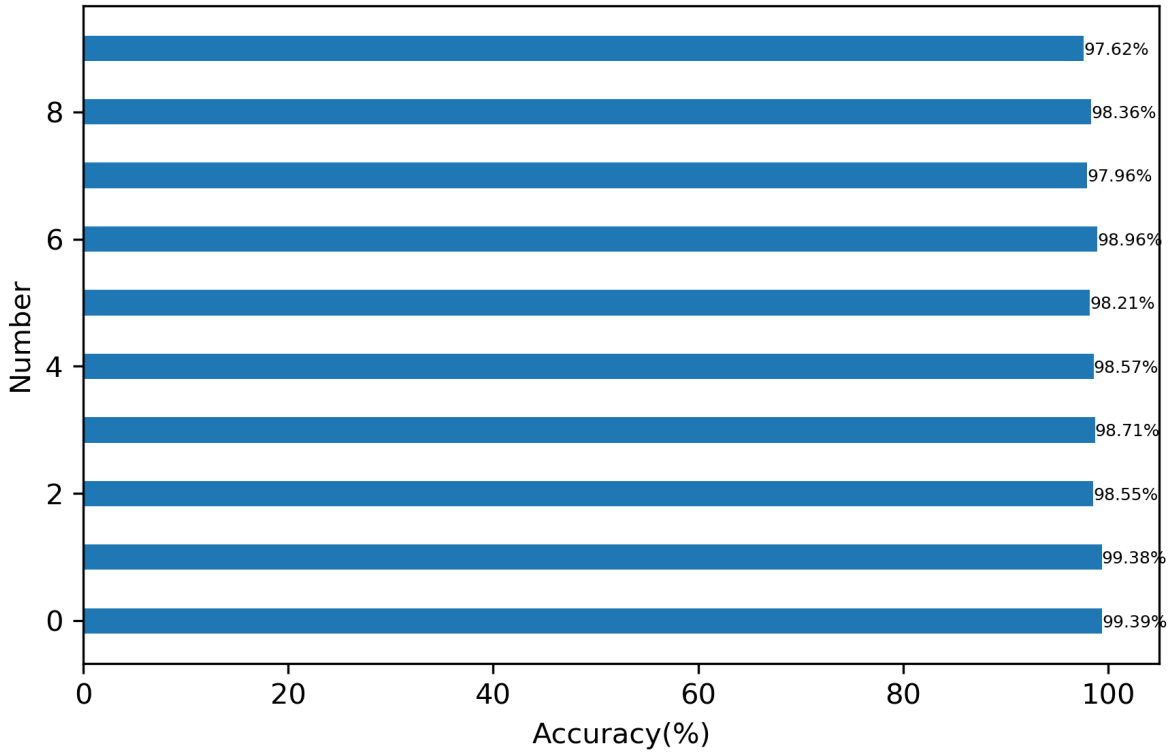



Fig6 不同数字识别准确率

根据Fig8 我们可以看到采用 RBF kernel的模型对于数字都有很高的识别准确率都在91%以上，对于0,1的识别准确率可以超过99%

为了进一步研究这些数字识别准确率低的原因，我们对识别结果做进一步分析，可以得到如下的识别错误矩阵如图 Fig7，矩阵元素（i, j）代表在训练样本中将数字 i 识别为 j 的次数。

| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Total Error |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 2 | 0 | 1 | 2 | 0 | 6 |
| 1 | 0 | 0 | 3 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 7 |
| 2 | 4 | 0 | 0 | 0 | 1 | 0 | 0 | 7 | 3 | 0 | 15 |
| 3 | 0 | 0 | 2 | 0 | 1 | 2 | 0 | 4 | 3 | 1 | 13 |
| 4 | 0 | 0 | 2 | 0 | 0 | 0 | 4 | 0 | 1 | 7 | 14 |
| 5 | 2 | 0 | 0 | 5 | 1 | 0 | 3 | 1 | 3 | 1 | 16 |
| 6 | 3 | 2 | 0 | 0 | 2 | 2 | 0 | 0 | 1 | 0 | 10 |
| 7 | 0 | 3 | 8 | 1 | 1 | 0 | 0 | 0 | 1 | 7 | 21 |
| 8 | 3 | 0 | 1 | 3 | 1 | 1 | 1 | 2 | 0 | 4 | 16 |
| 9 | 2 | 3 | 0 | 6 | 5 | 2 | 1 | 4 | 1 | 0 | 24 |

Fig7 识别误差矩阵

根据上面的图像，可以看出被误识别的整体概率很低，主要发生在一些比较相似的数字之间。如数字 2 被识别成 7， 数字 4 被识别成 9 等，通过仔细观察会发现数据中存在这些很难进行区分的数字，因此出现这样的现象也是正常的。

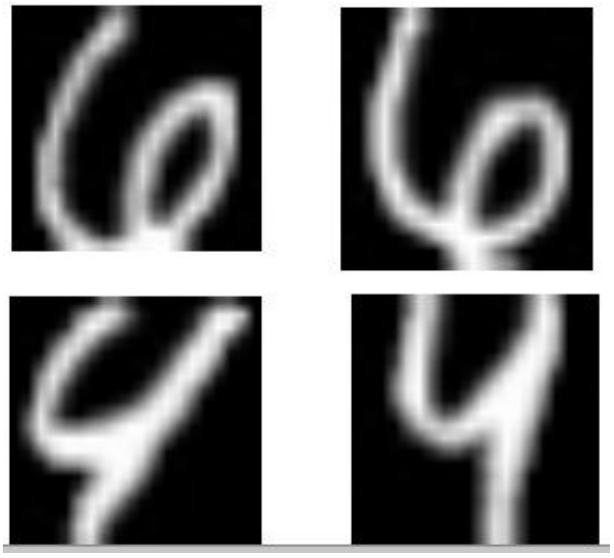


Fig 8 难以区分的数字

# 对比分析

前面，对两种模型都做了比较的详细地分析，这里讲两个模型做一个简单的对比，得到二者主要的优缺点等。

1. 识别的准确率是我们需要首先考量的，在相同的训练样本下可以得到二者是别的准确率如图Fig 9

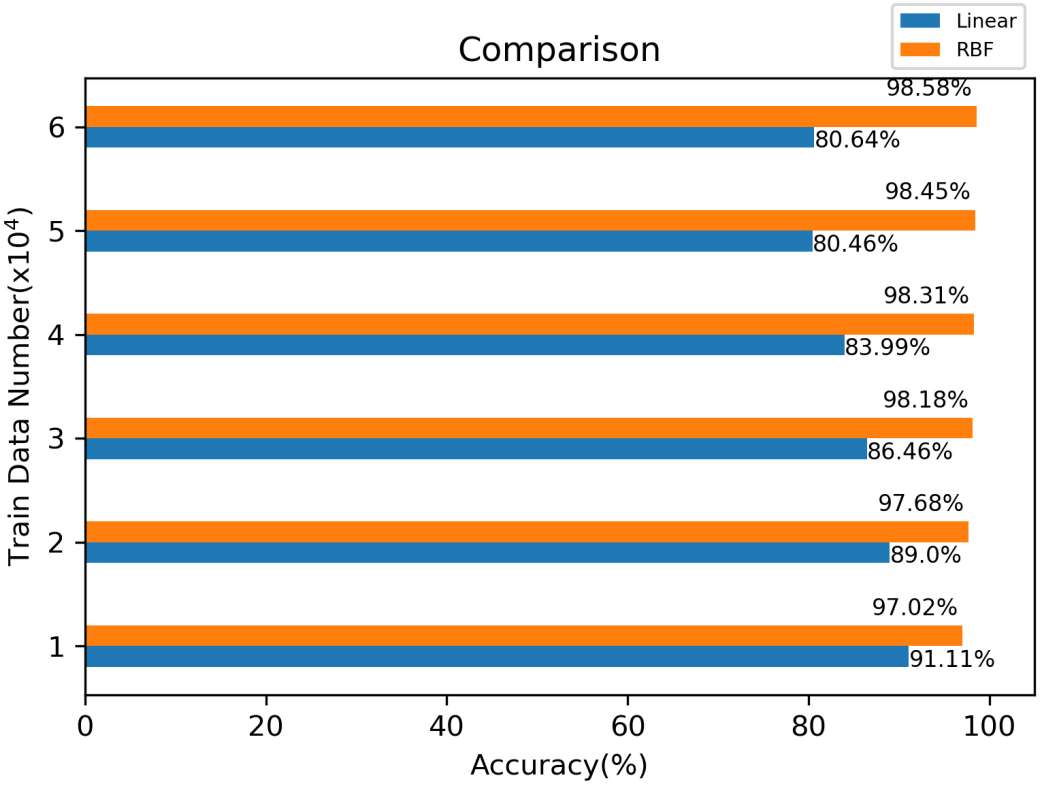



Fig 9识别准确率对比

可以看到在 RBF kernel 较 Linear kernel 有明显的优势,在各个训练集下都有更高的识别准确率。

2. 具体到对每一个数字识别的准确率，可以到如图Fig10 的对比图像

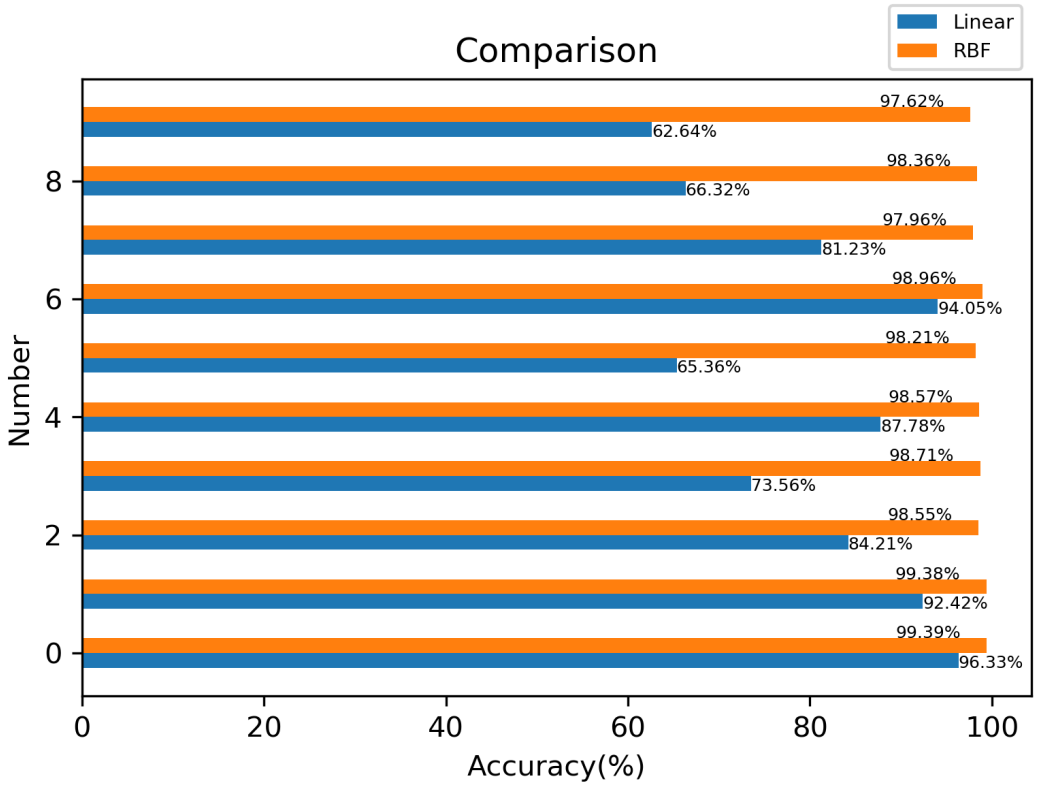



Fig10

可以看到在对具体数字的识别上，RGF kernel 对Linear kernel 也有碾压式的优势，在每一个数字上都变现出了很高的识别准确率。

3. 除了识别的准确率，时间性能也是需要考虑的一个因素

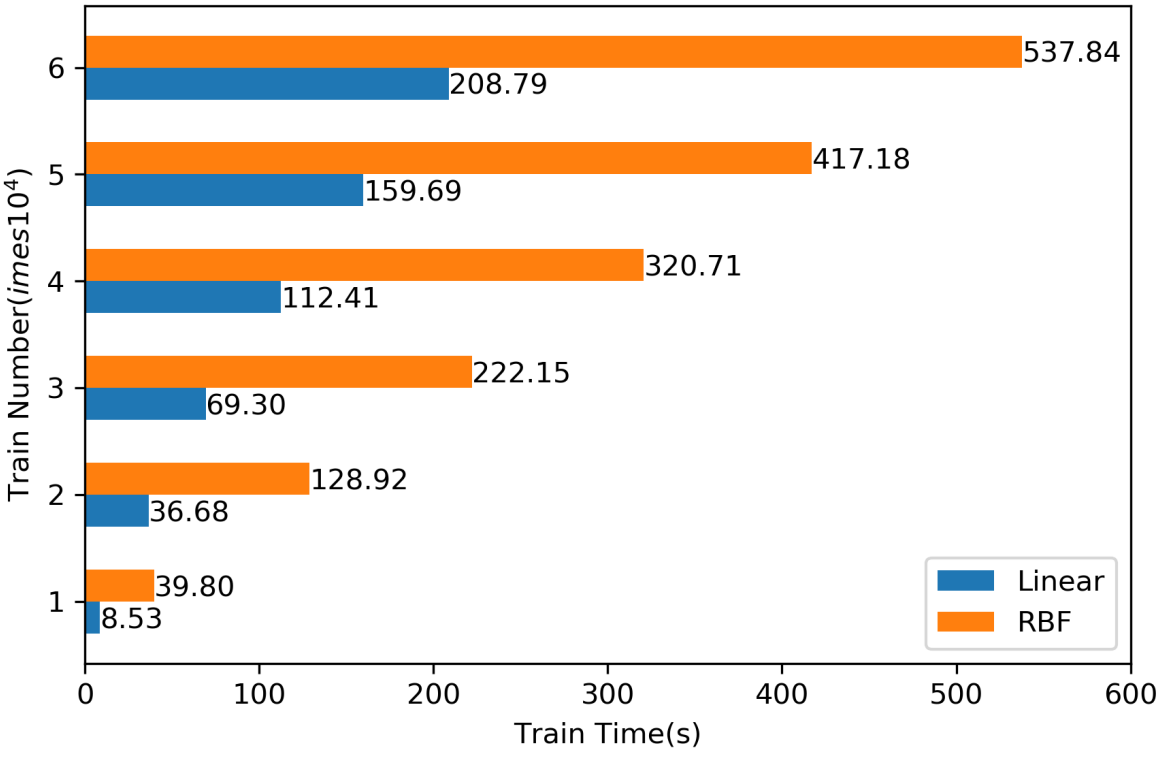
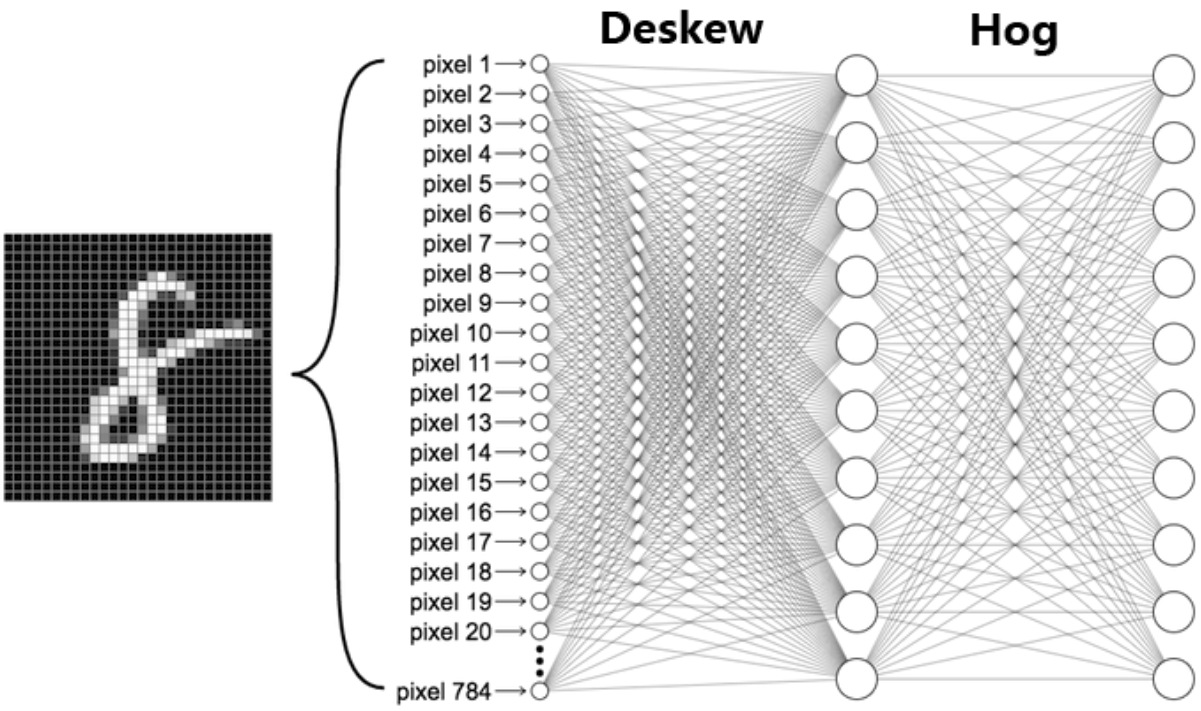



Fig11训练时间对比

# Hog 变换 + Linear kernel

HOG ： Histogram of Oriented Gradients 方向梯度直方图，是计算机视觉中常用的特征提取方法。就是计算图像在某一一个方向上的梯度值之后在进行累计得到直方图代表图像在这个方向上的特征。 在前面，Linear kernel 出现了过拟合的现象导致准确率下降。为了提高线性模型识别的准确率，对图像做HOG变换来提取图像的特征，减少训练集中向量的维度。之后利用提取出的特征来训练模型。




特征提取过程

# 图像修正

在提取Hog之前，先利用opencv中的函数对图形进行修正。

```
def deskw(img):
    img=np.reshape(img,[28,28])
    m = cv2.moments(img)
    if abs(m['mu02']) < 1e-2:
        return img.copy()
    skew = m['mu11']/m['mu02']
    M = np.float32([[1, skew, -0.5*SZ*skew], [0, 1, 0]])
    img = cv2.warpAffine(img,M,(SZ, SZ),flags=affine_flags)
    return img
```

下面是一个经过变换之后的图像如图Fig12

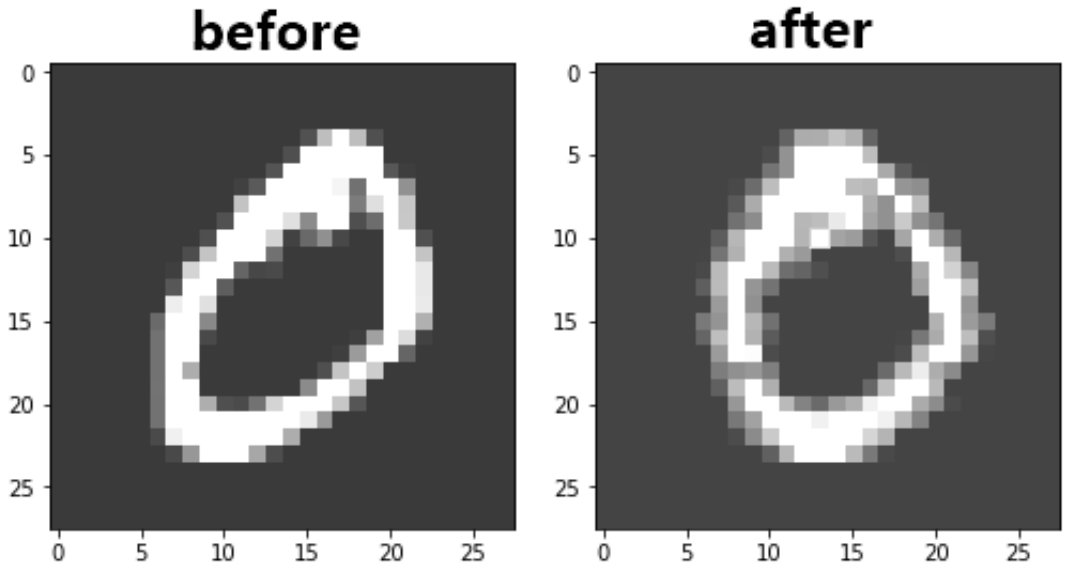



Fig12 矫正前后对比

可以发现矫正之后图像更符合正常的判断标准，这样有利于我们之后进行特征提取和训练。

# 提取Hog特征

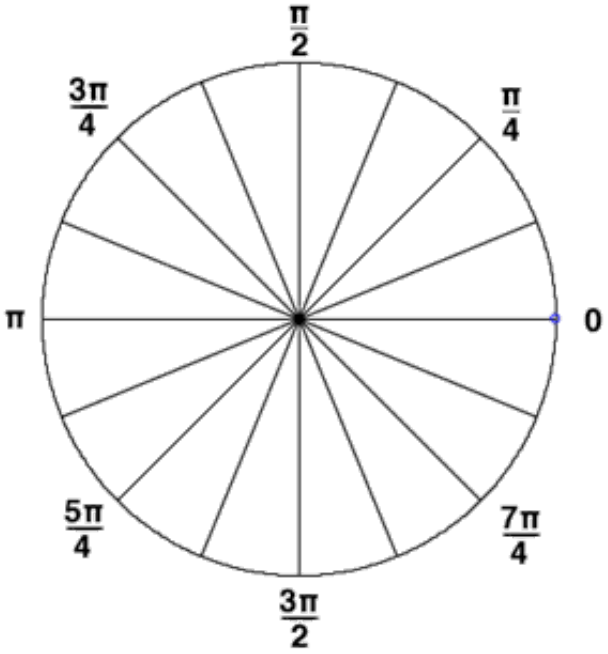
1. 先计算图像中每一个点在x，y方向上的梯度$G_x, G_y$，和对应的梯度的大小G:

$$G(x,y) = \sqrt{G_x^2 + G_y^2}$$

2. 计算该点梯度的方向$\theta$

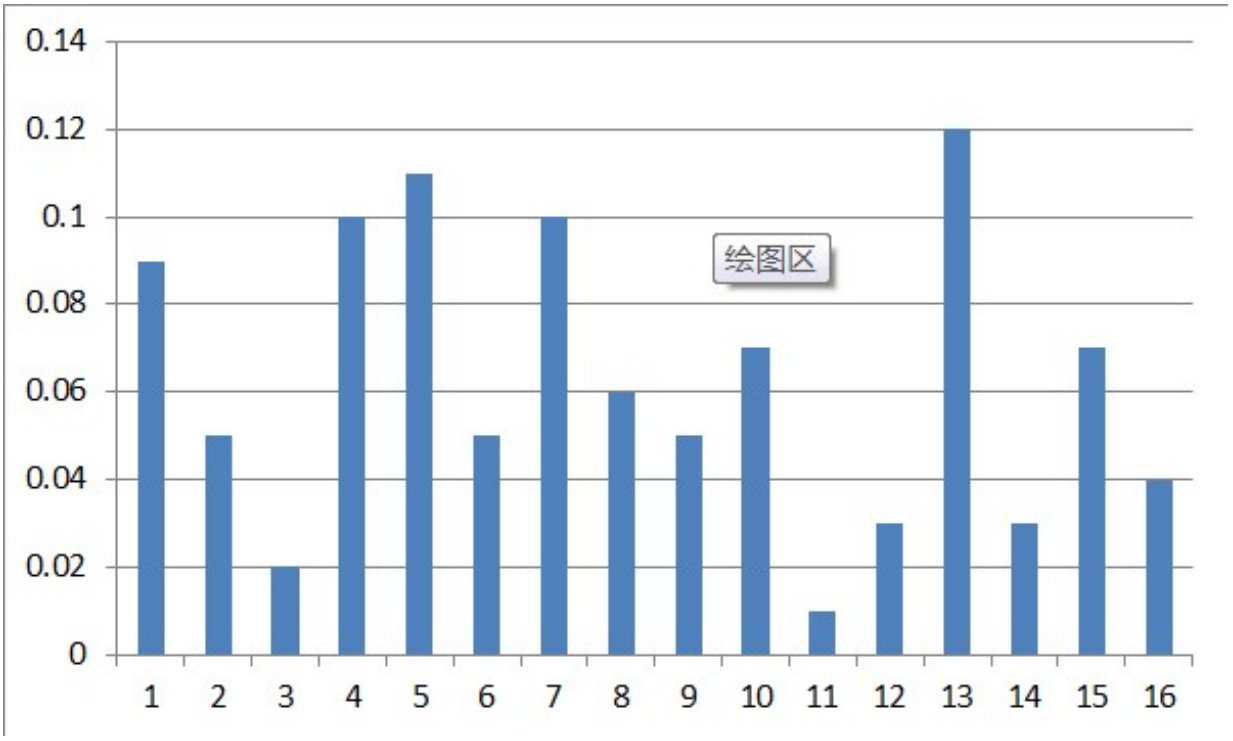
$$\theta(x,y) = arctan(\frac{G_x}{G_y})$$

3. 将图像按照角度进行分割，然后根据每个像素点的梯度方向，利用双线性内插法将其幅值累加到直方图中，计算在每个区域中的HOG




图像分割示意图

4. 统计所有区域中的 HOG 就可以得到整个图像的 HOG特征




HOG 特征统计

```
def hog(train_data):
    img=np.reshape(train_data,[28,28])

    # Define Sobel derivatives

    gx=cv2.Sobel(img,cv2.CV_32F,1, 0)
    gy=cv2.Sobel(img,cv2.CV_32F,0, 1)
    mag, ang=cv2.cartToPolar(gx,gy)

    # quantizing binvalues in (0...31)
        bin_n=31
        bins = np.int32(bin_n*ang/(2*np.pi))

     # Divide to 4 sub-squares
    bin_cells = bins[:14,:14], bins[14:,:14], bins[:14,14:], bins[1
4:,14:]
        mag_cells = mag[:14,:14], mag[14:,:14], mag[:14,14:], mag[1
4:,14:]
        # bincount(x,weight)
        # return_val[n]=number(x=n) ,如果有wieght的话对应的转化为+weigh
t

        hists = [np.bincount(b.ravel(), m.ravel(), (bin_n+1)) for
b, m in zip(bin_cells, mag_cells)]
        # reval(x,order='C'(F,K,A) 按照column, row等reshape成向量
        # bincout 中权重是 mag_cells 最小长度是 bin_cell
        hist = np.hstack(hists)
        hist=np.array(hist,dtype=np.float32)
        # 化成行向量
        return hist
```

# 模型分析

之后和之前一样先对参数进行格点搜索，然后对模型进行训练。因为搜索过程中发现模型对我们设置的参数不敏感，这里就直接采用默认的参数进行训练。和之前一样我们对模型的识别准确率和耗时等因素进行分析。

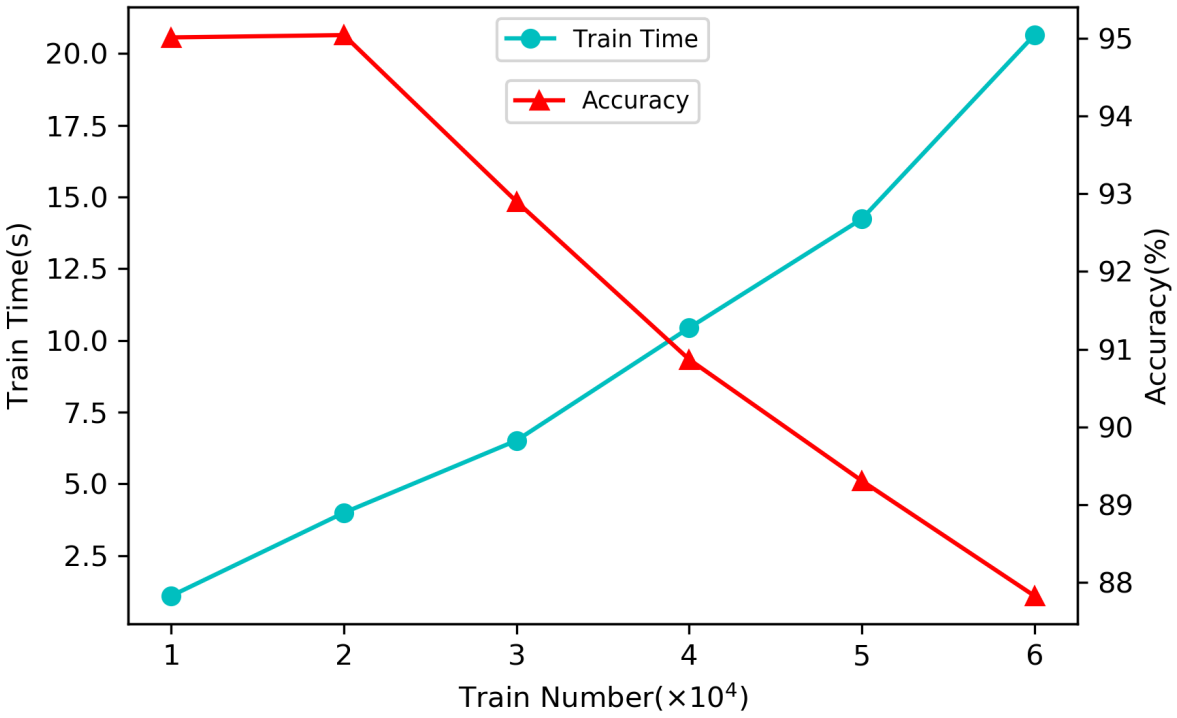



Fig 13

可以发现在Hog + Linear 模型中也存在随着样本增加识别率反而下降的现象。我认为这是因为模型中出现了过拟合的线性，之后我们会对次做详细的分析。

# 过拟合分析

可以发现在线性模型之中，随着样本数目的增加很容易出现过拟合的现象。为了证明这的确是过拟合现象，将训练集（60000）和测试集（10000）对调，也就是说用数量较少的测试集作为训练集，用数量较大训练集作为测试集。我们可以得到只使用 Linear kernel 以及使用 Linear kernel + Hog 的训练时间很准确率如下(T表示将训练集和测试集对调)

| | Train Time(s) | Accuracy(%) |
|---|---|---|
| Linear | 208.79 | 80.64 |
| Linear(T) | 14.75 | 90.69 |
| Linear+Hog(T) | 2.24 | 94.64 |

Fig 14

和之前对比我们可以发现此时在训练集缩小，测试集增大的情况下准确率得到了明显的提升，说明我们之前的模型中当训练样本较大的时候存在过拟合的现象，可以适当减少训练样本的数量消除过拟合。

最后我们对比一下在不过拟合的情况下线性模型对各个数字识别的准确率：

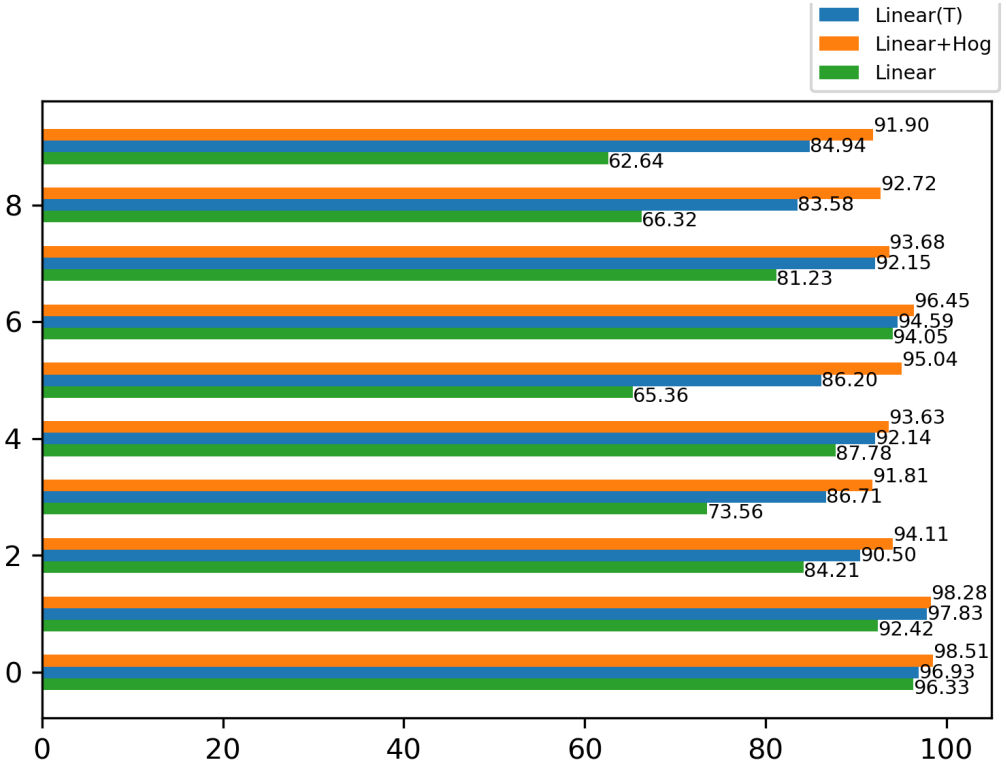



Fig15

根据上面的图像可以发现经过特征提取之后线性模型的对数字的识别能力都得到了增强。对每个数字识别准确率都在91%以上。除此之前根据Fig14 也可以发现模型的训练时间大大减小，因此选择合适的方法对原始数据进行特征提取不仅仅可以提高识别的准确率也可以缩小训练时间，大大提高模型的性能。