相关信息

学号：16337085

姓名：胡中林

班级：16智能科学与技术

作业使用了Jupyter Notebook编写，可以直接看ipynb文件

统计分析方法第三次作业

题目：使用PCA进行图像压缩

实验要求

> 输入一张灰度图片Lena，放大到256*256，使用PCA方法把原始图片分别按照2:1、8:1、32:1进行压缩，即压缩后的数据量为原始图片的1/2、1/8、1/32。分析压缩后的数据所含信息量大小，并比较压缩数据再经过重建后与原始图片的视觉差异。
>
> 把图像分割成很多块16*16，把每个小图像块看成不同的样本点，一个小图像块内每个像素是样本点的不同维度。
>
> 压缩率计算：16*16=256维度，压缩率为原始的1/2，即变成128维度。图片重新生成，是利用128维度重构图片块16*16，重构如上。、

实验原理

PCA简单来讲，就是把n个高维的压缩成n个低维的向量。PCA可以用于提取关键变量、或者是有损的压缩，PCA也是一种常用的降维方法。

以2:1的压缩率为例，实验流程如下：

1. 把Lena的图片分成很多块16*16，把每一个方块看成一个样本，把每个样本看成256维的向量
2. 用PCA算法把每个样本点从256维压缩到128维，把这个数据存好，和PCA的均值和其他信息都存好
3. 用PCA的均值和其他信息，把每个样本点从128维还原256维向量
4. 把每个样本点的256维向量还原成16*16的方块，并把方块拼接好，就是还原图像了
5. 然后比较不同的结果

实验过程

读入图片

```
import numpy as np
from scipy import ndimage
import matplotlib.pyplot as plt
from PIL import Image
from sklearn import decomposition
from scipy import linalg
```

```
original_image = ndimage.imread('原始图片.bmp')
```

测试一下自己写PCA和sklearn的PCA

先定义参数

```
block_size = 16
n_components = 8
```

sklearn里面的PCA

```
def ratio(n_components, block_size):
    return n_components / block_size ** 2

def CompresseImageSKL(original_image, block_size, n_components):
    original_block = []
    for x_start in range(0, original_image.shape[0], block_size):
        for y_start in range(0, original_image.shape[1], block_size):
            temp = original_image[x_start:x_start+block_size,y_start:y_start+block_size]
            original_block.append(temp.flatten())
#           print(x_start, y_start, len(original_block)-1)
    pca = decomposition.PCA(n_components=n_components)
    pca.fit(original_block)
    compressed_data = pca.transform(original_block)
    return compressed_data, pca

def DecompresseImageSKL(compressed_data, pca, block_size, n_components, image_shape, image_dtype):
    rebuild_block = pca.inverse_transform(compressed_data)
    rebuild_image = np.empty(image_shape, dtype=image_dtype)
    for x_start in range(0, image_shape[0], block_size):
        for y_start in range(0, image_shape[1], block_size):
            index = int(x_start*(image_shape[0]/block_size**2)+y_start/block_size)
    #         print(x_start, y_start, index)
            temp = rebuild_block[index]
            temp = temp.reshape((block_size, block_size))
            rebuild_image[x_start:x_start+block_size,y_start:y_start+block_size] = temp
    return rebuild_image
```

```python
compressed_data_skl, pca = CompresseImageSKL(original_image, block_size, n_components)
```

```python
rebuild_image_skl = DecompresseImageSKL(compressed_data_skl, pca, block_size, n_components, original_image.shape, original_image.dtype)
```

自己写的PCA

```python
# 把x降维到n_components维
def TransformPCA(x, n_components):
    mean = np.mean(x, axis = 0)
    x -= mean #归一化
    cov = np.cov(x, rowvar = False) #算方差
    evals, evecs = linalg.eigh(cov)
    idx = np.argsort(evals,)[::-1] #[::-1]是逆序
    evecs = evecs[:,idx]
    evals = evals[idx]
    result = np.dot(x, evecs)
    # 返回的结果不仅仅要用压缩的数据，还有有平均值等其他信息
    return result[:,:n_components], mean, evecs[:,:n_components].transpose((1,0))

# 把压缩完的数据还原到原来的空间中
def InverseTransformPCA(transformed_x, mean, components):
    result = np.dot(transformed_x, components) + mean
    return result
```

```python
def CompresseImageMine(original_image, block_size, n_components):
    original_block = []
    for x_start in range(0, original_image.shape[0], block_size):
        for y_start in range(0, original_image.shape[1], block_size):
            temp = original_image[x_start:x_start+block_size,y_start:y_start+block_size]
            original_block.append(temp.flatten())
#             print(x_start, y_start, len(original_block)-1)
    compressed_data, mean, components = TransformPCA(original_block, n_components)
    return compressed_data, mean, components

def DecompresseImageMine(compressed_data, mean, components, block_size, n_components, image_shape, image_dtype):
    rebuild_block = InverseTransformPCA(compressed_data, mean, components)
    rebuild_image = np.empty(image_shape, dtype=image_dtype)
    for x_start in range(0, image_shape[0], block_size):
        for y_start in range(0, image_shape[1], block_size):
            index = int(x_start*(image_shape[0]/block_size**2)+y_start/block_size)
    #         print(x_start, y_start, index)
            temp = rebuild_block[index]
            temp = temp.reshape((block_size, block_size))
            rebuild_image[x_start:x_start+block_size,y_start:y_start+block_size] = temp
    return rebuild_image
```

```
compressed_data_mine, mean, components = CompresseImageMine(original_image, block_size, n_components)
```

```
rebuild_image_mine = DecompresseImageMine(compressed_data_mine, mean, components, block_size, n_components, original_image.shape, original_image.dtype)
```

比较一下两个结果

sklearn里面的PCA压缩完的数据（compressed_data_skl）如下：

```
    array([[ 5.47891115e+02,  1.80701988e+01, -1.47846261e+01, ...,
              2.92474416e+00,  6.93805081e+00, -1.51620881e+01],
            [ 4.93071535e+02,  2.08687213e+01,  7.84115769e+00, ...,
              2.24578069e+00, -4.90866032e+00, -1.00276103e+01],
            [ 6.57395653e+02, -7.98980053e+01,  7.64529934e+00, ...,
             -1.21135484e+01, -9.22611483e+00, -1.88017700e+01],
            ...,
            [-4.00833210e+02, -2.75172149e+02,  1.71258086e+02, ...,
             -8.50753606e+01, -1.75960014e+01,  1.77691367e+01],
            [-5.92625912e+02,  3.47935159e+02, -1.94571072e+02, ...,
              2.24281039e+01,  2.90303622e+01,  1.50628433e+01],
            [-9.51385260e+02, -1.32151881e+02,  1.55709061e+02, ...,
              3.49835285e+01,  5.36024847e+00, -7.54452652e-01]])
```

自己写的PCA压缩完的数据（compressed_data_mine）如下：

```
    array([[-5.47891115e+02, -1.80701988e+01,  1.47846261e+01, ...,
             -2.92474258e+00,  6.93805088e+00,  1.51622465e+01],
            [-4.93071535e+02, -2.08687213e+01, -7.84115769e+00, ...,
             -2.24578283e+00, -4.90865931e+00,  1.00275660e+01],
            [-6.57395653e+02,  7.98980053e+01, -7.64529934e+00, ...,
              1.21135456e+01, -9.22611652e+00,  1.88016834e+01],
            ...,
            [ 4.00833210e+02,  2.75172149e+02, -1.71258086e+02, ...,
              8.50753598e+01, -1.75960006e+01, -1.77692915e+01],
            [ 5.92625912e+02, -3.47935159e+02,  1.94571072e+02, ...,
             -2.24281044e+01,  2.90303598e+01, -1.50627753e+01],
            [ 9.51385260e+02,  1.32151881e+02, -1.55709061e+02, ...,
             -3.49835323e+01,  5.36025001e+00,  7.54346863e-01]])
```

sklearn里面的PCA重构的图片矩阵元素（rebuild_image_skl）如下：

```
    array([[159, 161, 161, ..., 152, 150, 145],
           [159, 161, 162, ..., 150, 147, 139],
           [159, 161, 162, ..., 148, 143, 135],
           ...,
           [ 47,  49,  52, ..., 100, 103, 104],
           [ 49,  51,  52, ..., 103, 105, 106],
           [ 49,  52,  54, ..., 106, 106, 107]], dtype=uint8)
```

自己写的PCA重构的图片矩阵元素（rebuild_image_mine）如下：

```
    array([[159, 161, 161, ..., 152, 150, 145],
           [159, 161, 162, ..., 150, 147, 139],
           [159, 161, 162, ..., 148, 143, 135],
           ...,
           [ 47,  49,  52, ..., 100, 103, 104],
           [ 49,  51,  52, ..., 103, 105, 106],
           [ 49,  52,  54, ..., 106, 106, 107]], dtype=uint8)
```

可以看到，我自己写的PCA和sklearn里面的PCA压缩之后的数据是一样的，只不过恰好互为相反值，但是这个其实没什么影响，毕竟找到的主成分都是一样的。而且，两种方法重构出来的图片都是一样的。证明自己写的PCA没错。

```
plt.imshow(original_image, cmap='gray')
plt.show()
plt.imshow(rebuild_image_skl, cmap='gray')
plt.show()
plt.imshow(rebuild_image_mine, cmap='gray')
plt.show()
```

用自己写的PCA分别按照2:1、8:1、32:1进行压缩

```
def compute_n_components(ratio, block_size):
    return int(block_size**2 / ratio)
```

```
for ratio in [2, 8, 32, 128]:
    n_components = compute_n_components(ratio, block_size)
    compressed_data, mean, components = CompresseImageMine(original_image, block_size, n_components)
    rebuild_image = DecompresseImageMine(compressed_data, mean, components, block_size, n_components, original_image.shape, original_image.dtype)
    print('ratio = ', 1 / ratio)
    plt.imshow(rebuild_image, cmap='gray')
    plt.show()
    Image.fromarray(rebuild_image).save('ratio_%d.bmp'%ratio)
```

原图就可以看成是压缩比是1:1的情况

```
Image.fromarray(original_image).save('ratio_1.bmp')
```

压缩比是1:1，就是原图：

压缩比是2:1，即ratio = 0.5时候的效果：

压缩比是8:1，即ratio = 0.125时候的效果：

压缩比是32:1，即ratio = 0.0078125时候的效果：

实验结果分析

可以看到，随着压缩比的增大，图片越来越不清晰，细节也损失得越来越多了。当2:1的时候，图片基本上没太大的变化，除了头发这些细节，基本上和原图没区别。当8:1的时候明显可以看出来，图片被压缩了，头发、眼睛等地方和原图有明显的区别，但是背景、大面积的皮肤这些地方还没有明显的变化。当32:1的时候，图片信息丢失十分严重，连个圆形的眼珠都找不到了，仅有背景能大概保持原样，而且能够明显看到每个方块之间的边界。这说明了图片有比较多的冗余信息，而且当压缩的时候，细节多的地方先出现偏差，细节少的地方变化不大。

相关问题

1. 在压缩的时候，压缩完的向量不应该用浮点型存储的（浮点比较耗空间），压缩完的向量应该要用图片举证元素的相同格式存（uint8），这样的压缩在存储空间上才比较像样。
2. 在实际使用当中，不仅仅要存好压缩之后的数据（就是128维的数据），还要存放PCA的均值和其他信息，而这些信息也需要一定的空间，所以实际的压缩效率没那么高。