

主成分分析

从一个例子慢慢讲起

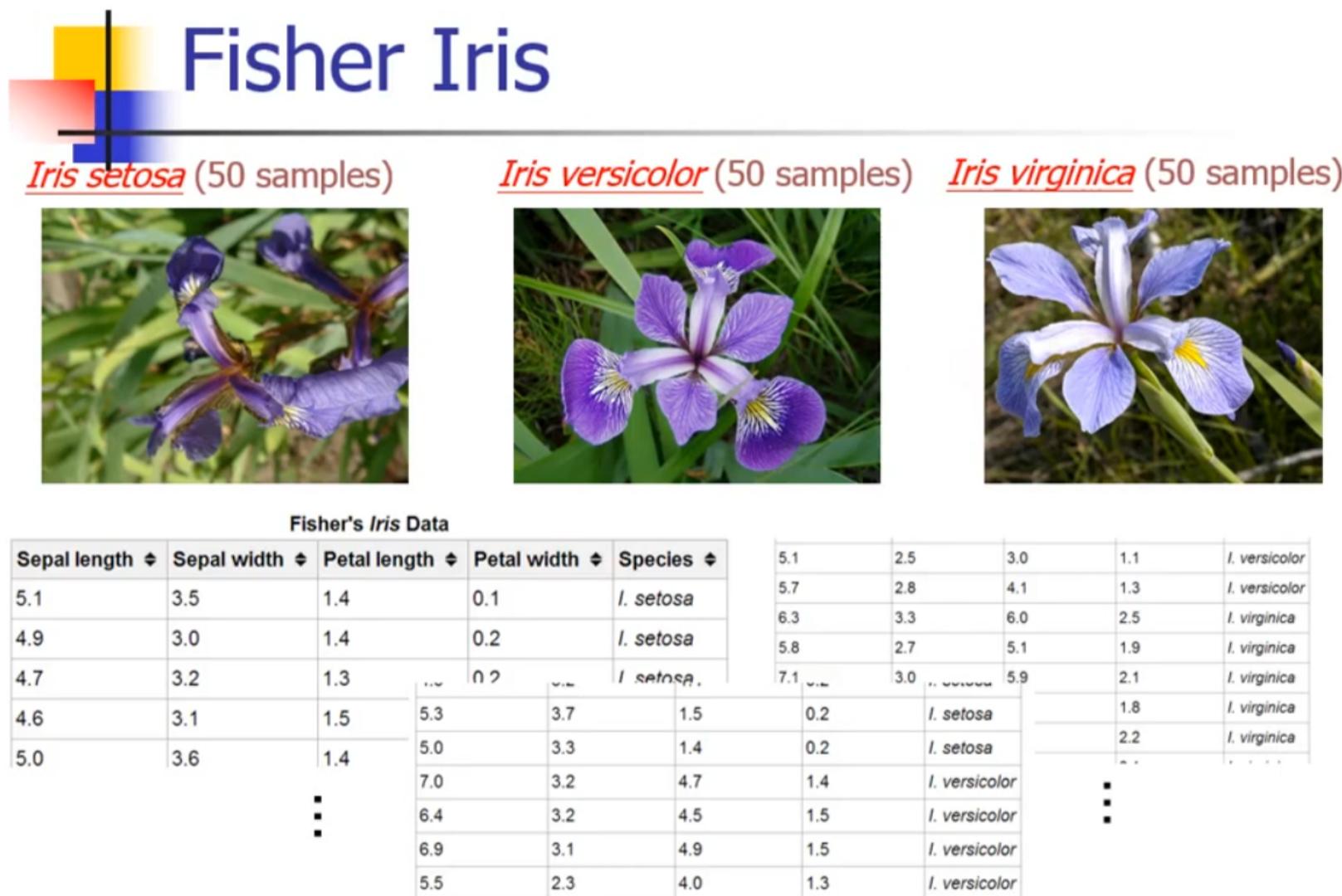
```
In [1]: import numpy as np  
        import pandas as pd  
        import torch  
        import matplotlib.pyplot as plt  
        from PIL import Image
```

关于涉及到的一些乘法运算

二维矩阵乘法和点乘归纳

	矩阵乘法	逐元素点乘
numpy	<code>np.dot()</code> <code>np.matmul()</code>	*
torch	<code>torch.mm()</code> <code>torch.matmul()</code>	<code>torch.mul</code> , *
pandas	<code>a.dot(b)</code>	*

鸢尾花数据集简单介绍



数据集包含150个样本，第1-50个样本是setosa，第51-100是versicolor，第101-150是virginica。

其中每个样本使用4个特征来表示，分别是花萼长度、花萼宽度、花瓣长度、花瓣宽度。

```
In [2]: data=pd.read_csv("./data/Iris.csv", header=0, index_col="Id")
        data=data.iloc[:, :4]
        setosa=data.iloc[:50, :]
        versicolor=data.iloc[50:100, :]
        virginica=data.iloc[100:150, :]
        setosa.head(), versicolor.head(), virginica.head()
```

```
pca
Out[2]: (   SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
   Id
1      5.1          3.5          1.4          0.2
2      4.9          3.0          1.4          0.2
3      4.7          3.2          1.3          0.2
4      4.6          3.1          1.5          0.2
5      5.0          3.6          1.4          0.2,
   SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
   Id
51     7.0          3.2          4.7          1.4
52     6.4          3.2          4.5          1.5
53     6.9          3.1          4.9          1.5
54     5.5          2.3          4.0          1.3
55     6.5          2.8          4.6          1.5,
   SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
   Id
101    6.3          3.3          6.0          2.5
102    5.8          2.7          5.1          1.9
103    7.1          3.0          5.9          2.1
104    6.3          2.9          5.6          1.8
105    6.5          3.0          5.8          2.2)
```

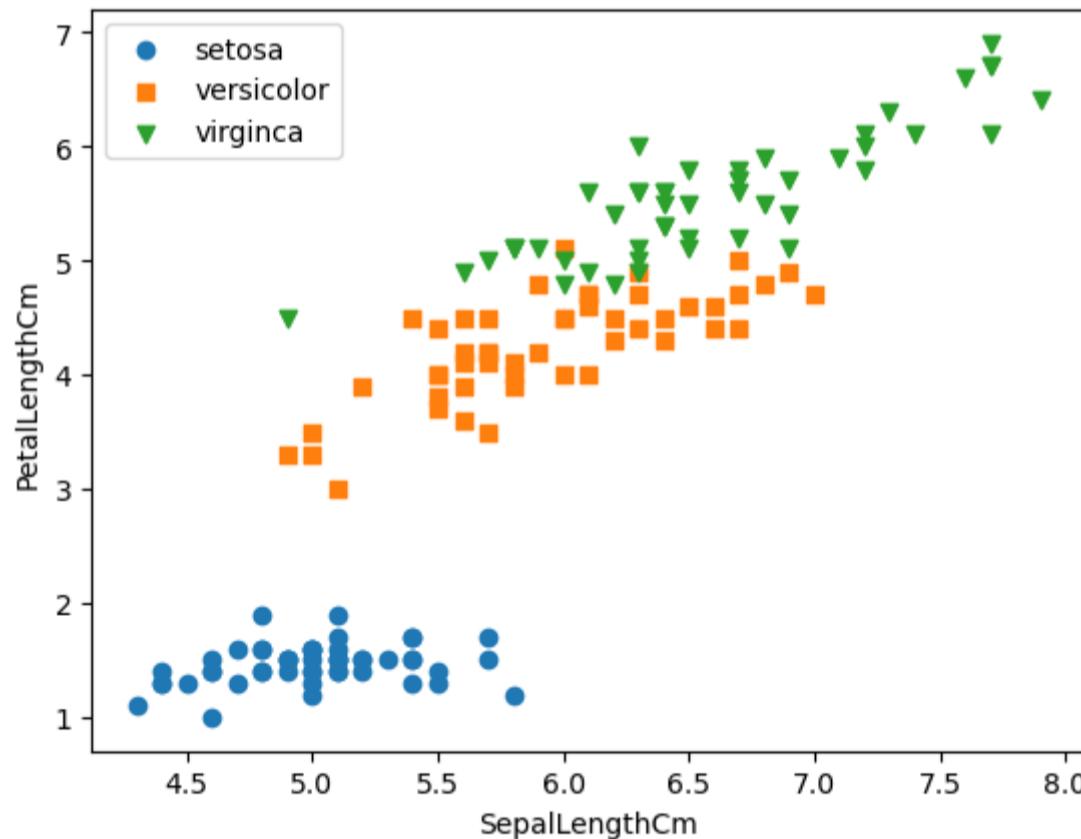
在二维平面可视化展示

对于上面这150条数据，我们首先不拿它们作任何处理，随便先取两个特征，在一个二维坐标平面上先展示出来，让大家有个直观认识

```
In [3]: setosa_x=setosa.loc[:, "SepalLengthCm"]           # 拿setosa的花萼长度作为横坐标
setosa_y=setosa.loc[:, "PetalLengthCm"]                 # 拿setosa的花瓣长度作为纵坐标
versicolor_x=versicolor.loc[:, "SepalLengthCm"]
versicolor_y=versicolor.loc[:, "PetalLengthCm"]
virginica_x=virginica.loc[:, "SepalLengthCm"]
virginica_y=virginica.loc[:, "PetalLengthCm"]
```

```
In [4]: plt.scatter(setosa_x, setosa_y, marker="o", label="setosa")      # setosa的图形使用圆形表示
plt.scatter(versicolor_x, versicolor_y, marker="s", label="versicolor")
plt.scatter(virginica_x, virginica_y, marker="v", label="virginica")
plt.xlabel("SepalLengthCm")
plt.ylabel("PetalLengthCm")
plt.legend()                                              # 显示设置的label图例
```

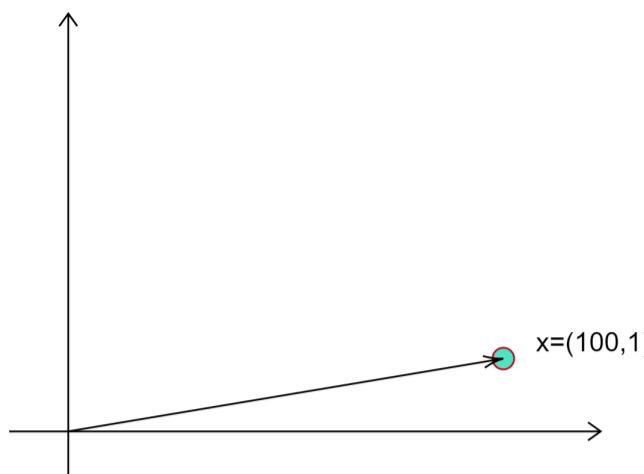
Out[4]: <matplotlib.legend.Legend at 0x27555bc0790>



那么我们要看上面这张图的什么东西呢，最直观的感觉就是三种花很具有区分性，随便给图上的一个坐标 (SepalLengthCm, PetalLengthCm)，估计你都能大致说出这朵花是哪种。对于上面这个过程，我们做了一件很重要的事，就是舍弃了这些花的部分特征，只保留两个特征，我们比较幸运，保留的这两个特征还是抓住了这三种花的本质，虽然丢弃了一些特征，但是丢失的信息貌似并不多。但是如果数据的特征维度有1000维，或许我们闭着眼睛去选取特征，丢弃特征，就没那么容易了，很容易损失这批数据的主要信息。

有了上面的分析，我们初步的目标就比较明确了，如何丢弃一批数据的特征，达到降维的效果，并且还不能丢失这批数据的大部分信息。而我们上面的分析比较直观，仅仅靠能不能区分来看我们保留的特征是不是主要信息，进而判断丢失的信息多不多。接下来我们从定性计算的角度去分析如何去丢掉数据的信息，保留重要信息，达到“数据压缩”的目的，这就是我们这节要讲的第一件事。

数据压缩的几个简单例子



对于上图中的向量 $x = (100, 1)$, 我们简单粗暴把 x 的长度 d 作为它所包含的信息量, 那么 $information_capacity(x) = 100^2 + 1^2$, (简单起见, 长度平方一下)
然后从图上不难看出, 这个向量在纵轴方向上几乎没有什么信息量, 不妨舍弃。
那么就有 $information_capacity_compress(x) = 100^2$, 那么两者的信息只差1

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos nx + b_n \sin nx)$$

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos nx dx$$

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin nx dx$$

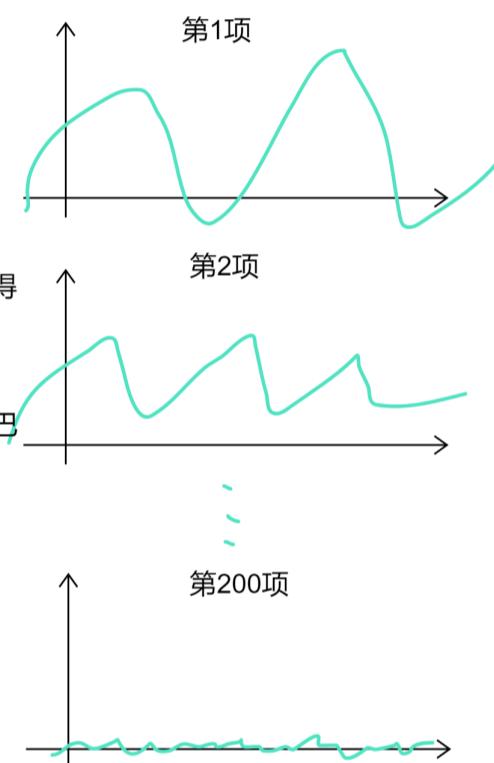
$\|f\| =$ 所有系数的平方和, 由推广形式的勾股定理可得

再由黎曼勒贝格引理可知, 当 $n \rightarrow \infty$, $a_n, b_n \rightarrow 0$

所以 n 取到 100, 200, 300 等有限数的时候就够了, 尾巴

的部分太小了, 统统丢了, 右边画一个演示图,

把构成 $f(x)$ 的所有项画出来, 可以看出, 越往后面
项越小, 后面的那些项相当于上面三角形里面比较
短的直角边, 可以看成杂讯, 飞机往 x 轴飞行, y 轴
有点风, 把飞机飞行方向刮偏移了一点点。



上面的例子在小波分析里比较常见, 从函数空间里找各种基向量, 然后函数就能有个表示, 函数版的线性代数就有了, 函数空间里的函数变换就那么

几个, 卷积, 拉普拉斯, 只不过其表示不是矩阵, 而是积分, 描述矩阵形式的变换的时候会拿几个特征数, 表示这个变换其实就是伸缩旋转
(算子的范数就是伸缩倍数, 信号放大率) 在描述函数空间里面的各种变换的时候, 用的就是谱, 也是一堆数字。不过这些东西有它的用途,
可能是解方程, 应用于控制论方向, 也可能是别的用途。我们这里只要了解它可以分解成一堆东西来表示即可,
差不多就是笛卡尔坐标系的推广, 谱系, 至于这个谱系上面的刻度怎么定义, 不需要知道。

上面的闲谈只是让你从别的角度了解一下我们讲的东西, 以及跟我们现在讲的角度做个区分,

除了上面两个例子, 我们前面还讲了一个例子, 就是矩阵的奇异值分解, 为什么它也能看成一种数据压缩呢?

虽然前面分析里面提到过, 这里再单独拎出来再说一遍

对于一般的 $m \times n$ 矩阵 A , 设 $rank A = r$, 那么有奇异值分解:

$$A = U \Sigma V^T = [u_1 \ u_2 \ \dots \ u_r \ \dots \ u_m] \begin{bmatrix} \sigma_1 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 0 & \sigma_3 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots \\ 0 & 0 & 0 & \cdots & \sigma_r & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} v_1^T \\ v_2^T \\ v_3^T \\ \vdots \\ v_r^T \\ v_{r+1}^T \\ \vdots \\ v_n^T \end{bmatrix} = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \cdots + \sigma_r u_r v_r^T$$

其中, $\sigma_i = \sqrt{\lambda_i}$, $u_i = Av_i / \sigma_i$, v_i 是 $A^T A$ 对应特征值 λ_i 的特征向量, $1 \leq i \leq r$.

数据压缩(秩k重建):

令 $\tilde{A} = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \cdots + \sigma_k u_k v_k^T$, $1 \leq k \leq r$. 说明:

(1). 保存矩阵 \tilde{A} 只需要 k 个奇异值, k 个 $m \times 1$ 向量, k 个 $n \times 1$ 向量, 总共需要 $k(m+n+1)$ 个数, 而保存 A 需要 $m \times n$ 个数,

假如 $m=n=1000$, $k=100$, \tilde{A} 需要 $100 \times (1000+1000+1) = 200100$ 个数, A 需要 $1000 \times 1000 = 1000000$ 个数, 节省了80%的存储空间

(2).把矩阵A看成向量, 考虑向量的2范数, 也就是矩阵的F范数, 那么秩k重建的矩阵与原来的矩阵的F范数距离为:

$$\|A - \tilde{A}\|_F = (\sigma_{k+1}^2 + \sigma_{k+2}^2 + \dots + \sigma_r^2)^{\frac{1}{2}}$$

(3).根据上面, 可以取前k个奇异值, 使其平方和占全部的95%, 从而保证信息丢失的不多

(4).如果这里A是一张图片, 那么 \tilde{A} 与原图A的每个pixel作差再平方, 求平方差损失, 这个平方差损失就是上面丢失的信息, 是一个很小的量

回头再看鸢尾花数据集的数据压缩

这里我们提供两个思路, 第一个思路就是奇异值分解, 第二个思路就是PCA

奇异值分解:

我们如果直接对上面150个样本的数据 $A(150 \times 4)$ 作奇异值分解, 再取前几个大的奇异值, 把 A 压缩为 \tilde{A} ,

我们需要思考一个问题, 那就是这样处理后, 每一行还是原来的每朵花吗? 这个问题我们暂时留下来, 等PCA讲完后再来解释

PCA:

PCA的数据压缩(降维)思路主要参考上面3个例子中的前两个, 就是向量相当于三角形, 丢掉短的那个直角边,

由于我们这里每个样本都是4维向量, 所以直角边有4条, 但是观察后发现, 对于不同样本, 短的那个直角边是不一样的,

我们如果丢掉比如花萼长度这个边, 发现不太合理, 里面有长有短, 对于某一朵花来说丢掉的信息可能多可能少, 所以直接丢某个特征会损失很多东西

这时候我们采用另一个思路, 就是换个参考坐标系, 当前坐标系的基向量是 $[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]$, 然后上面的数据就是在这组基向量下的坐标表示, 我们现在换组基向量, 第一个想法就是把坐标原点移动到平均值上面,

基向量变成 $\bar{x} + [1, 0, 0, 0], \bar{x} + [0, 1, 0, 0], \bar{x} + [0, 0, 1, 0], \bar{x} + [0, 0, 0, 1]$. 这样每个样本都要减去这些样本的平均, 得到新的坐标

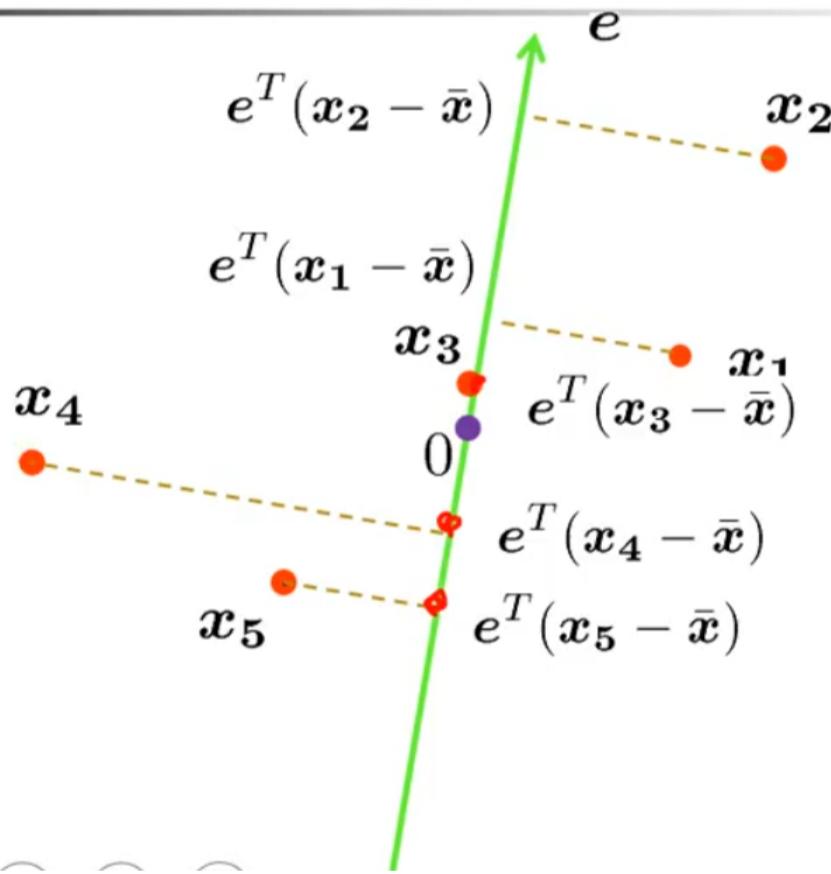
表示, 然后基于新的坐标表示考虑丢掉一些特征, 这样考虑的原因是平均信息一般比较重要, 一般不丢掉, 但是即便是这样, 还是存在上面所说的问题, 于是

在此基础上, 我们再作调整, 把新坐标轴旋转一些角度, 调整到这些样本的信息在某个轴上的投影分量(代表这批样本的信息量, 能量)最大, 而在另外一些轴

上的分量总和比较小, 小到随便丢掉也不至于损失太多信息, 设新的基向量是 e_1, e_2, e_3, e_4 , 基于这个考虑, 我们把每个样本投影到 e_1 上, 希望在 e_1

上的投影分量最大, 由于 e_1 是待求的, 我们设为 e

Principal Component Analysis



投影到 e 上的能量总和为:

$\sum_{i=1}^n (e^T(x_i - \bar{x}))^2$, 最大化这个数值, 等价于最大化这些样本投影到 e 上的方差:

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n (e^T(x_i - \bar{x}))^2$$

也就是我们要找的轴, 它既是所有样本去除均值向量后, 包含全体样本能量最强的地方, 也是这些样本散的最开的一个方向, 方差最大的方向

公式化简与求解:

$$\begin{aligned}\sigma^2 &= \frac{1}{n} \sum_{i=1}^n (\mathbf{e}^T (\mathbf{x}_i - \bar{\mathbf{x}}))^2 = \frac{1}{n} \sum_{i=1}^n (\mathbf{e}^T (\mathbf{x}_i - \bar{\mathbf{x}})) (\mathbf{e}^T (\mathbf{x}_i - \bar{\mathbf{x}}))^T \\ &= \frac{1}{n} \sum_{i=1}^n (\mathbf{e}^T (\mathbf{x}_i - \bar{\mathbf{x}})) ((\mathbf{x}_i - \bar{\mathbf{x}})^T \mathbf{e}) \\ &= \mathbf{e}^T \left(\frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i - \bar{\mathbf{x}}) (\mathbf{x}_i - \bar{\mathbf{x}})^T \right) \mathbf{e} = \mathbf{e}^T \Sigma \mathbf{e}\end{aligned}$$

其中 Σ 是原先那堆样本的协方差矩阵

$$\begin{aligned}\Sigma &= \frac{1}{n} \sum_{i=1}^n \begin{bmatrix} \sigma_{11} & \sigma_{12} & \dots & \sigma_{1p} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{p1} & \sigma_{p2} & \dots & \sigma_{pp} \end{bmatrix} = \frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i - \bar{\mathbf{x}}) (\mathbf{x}_i - \bar{\mathbf{x}})^T \\ &= \frac{1}{n} \sum_{i=1}^n \begin{bmatrix} (x_{i1} - \bar{x}_1)(x_{i1} - \bar{x}_1) & (x_{i1} - \bar{x}_1)(x_{i2} - \bar{x}_2) & \dots & (x_{i1} - \bar{x}_1)(x_{ip} - \bar{x}_p) \\ (x_{i2} - \bar{x}_2)(x_{i1} - \bar{x}_1) & (x_{i2} - \bar{x}_2)(x_{i2} - \bar{x}_2) & \dots & (x_{i2} - \bar{x}_2)(x_{ip} - \bar{x}_p) \\ \vdots & \vdots & \ddots & \vdots \\ (x_{ip} - \bar{x}_p)(x_{i1} - \bar{x}_1) & (x_{ip} - \bar{x}_p)(x_{i2} - \bar{x}_2) & \dots & (x_{ip} - \bar{x}_p)(x_{ip} - \bar{x}_p) \end{bmatrix}\end{aligned}$$

其中，协方差矩阵的对角线元素，就是这些样本每个特征的方差，其他元素是两个特征之间的协方差，如果把这堆样本的数据减去平均后的矩阵记为 X ,那么 $\Sigma = \text{Gram}(X) = X^T X$

好，暂时忘记协方差矩阵，后面还会对它做一些分析，我们现在回到求解:

$$e_1 = \underset{\mathbf{e}}{\operatorname{argmax}} \sigma^2 = \underset{\mathbf{e}}{\operatorname{argmax}} \mathbf{e}^T \Sigma \mathbf{e}, \text{ subject to: } \mathbf{e}^T \mathbf{e} = 1$$

Lagrange function

$$f(\mathbf{e}, \lambda) = \mathbf{e}^T \Sigma \mathbf{e} + \lambda (1 - \mathbf{e}^T \mathbf{e})$$

求导:

$$\begin{aligned}\frac{\partial f}{\partial \mathbf{e}} &= 2 \sum \mathbf{e} + \lambda (0 - 2\mathbf{e}) = 0 \\ 2 \sum \mathbf{e} - 2\lambda \mathbf{e} &= 0 \\ \sum \mathbf{e} &= \lambda \mathbf{e}\end{aligned}$$

因此 e_1 是协方差矩阵 Σ 的特征向量，所有样本在 e_1 上投影的方差值为: $e_1^T \Sigma e_1 = e_1^T \lambda_1 e_1 = \lambda_1$

由于特征值和特征向量不止一个，所以主轴不止一个，我们按投影在这些主轴上的方差大小（方差就是对应的 Σ 的特征值）排序，分别称为第一主轴 e_1 ，第二主轴 e_2 ，第三主轴 e_3 ...第 p 主轴 e_p ,其中 p 是协方差矩阵的边长，并且协方差矩阵一定有 p 个标准正交特征向量（回顾线性代数知识）

现在我们把每个样本投影到新的坐标轴上，求解新的坐标表示: $[y_1, y_2, \dots, y_p]$

$$\begin{aligned}\mathbf{y} &= \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1^T (\mathbf{x} - \bar{\mathbf{x}}) \\ \mathbf{e}_2^T (\mathbf{x} - \bar{\mathbf{x}}) \\ \vdots \\ \mathbf{e}_p^T (\mathbf{x} - \bar{\mathbf{x}}) \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \\ \vdots \\ \mathbf{e}_p^T \end{bmatrix} (\mathbf{x} - \bar{\mathbf{x}}) \\ &= \mathbf{A}_p^T (\mathbf{x} - \bar{\mathbf{x}})\end{aligned}$$

where $\mathbf{A}_p = [\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_p] \in R^{p \times p}$

由于我们是按这堆样本在轴上投影的能量从大到小排序的，所以最后几个轴上剩余的能量不多，我们不妨丢掉，取前 t 个主轴

$$\begin{aligned}\hat{\mathbf{y}} &= \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_t \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1^T (\mathbf{x} - \bar{\mathbf{x}}) \\ \mathbf{e}_2^T (\mathbf{x} - \bar{\mathbf{x}}) \\ \vdots \\ \mathbf{e}_t^T (\mathbf{x} - \bar{\mathbf{x}}) \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \\ \vdots \\ \mathbf{e}_t^T \end{bmatrix} (\mathbf{x} - \bar{\mathbf{x}}) \\ &= \mathbf{A}_t^T (\mathbf{x} - \bar{\mathbf{x}})\end{aligned}$$

where $\mathbf{A}_t = [\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_t] \in R^{p \times t}, t \leq p$

重建:

既然在新坐标轴下我们丢掉了部分信息，达到了降维或者数据压缩的目的，降维后我们其实可以进行可视化分析，或者别的分析，但是进行别的分析之前，我们先给出重建公式，即把坐标还原回去，得到原坐标系下的表示

(先把上面准确值求出来，在准确值公式中， A_p^T 求逆拿到 y 左边，其实就是取转置，然后把公式里面的 x, y 替换为估计值即可)

$$y = A_p^T (\mathbf{x} - \bar{\mathbf{x}}) \Rightarrow \mathbf{x} - \bar{\mathbf{x}} = A_p y \Rightarrow \mathbf{x} = \bar{\mathbf{x}} + A_p y \approx \bar{\mathbf{x}} + A_t \hat{y} = \hat{\mathbf{x}}$$

其中， A_p 形状 $p \times p$, A_t 形状 $p \times t$, \hat{y} 形状 $t \times 1$, $\bar{\mathbf{x}}$ 形状 $p \times 1$, 原始特征维度为 p

即最终重建公式为:

$$\hat{x} = \bar{x} + A_t \hat{y}$$

另外从这里可以看出重建的特征维度还是老样子，所以降维体现在主轴上面的降维，数据损失的分量也是变换到主轴上分析的，还原重建后就别作差分析损失了多少了

停下来休息一会儿

前面咕噜咕噜讲了一大堆，算了一大堆，加上后面还会分析很多东西，这里我们稍微整理一下前面讲了哪些东西

其一，我们上文围绕一条主线分析的，就是把每个样本看成一个三角形的斜边，其长度就是其能量，希望找到新的坐标系，使得所有样本在新坐标系下，删掉最后几条直角边，也不至于丢失太多能量。

新坐标系下的坐标表示，我们并没有过多分析，pca不可能仅仅就是换个坐标表示降个维就结束了，

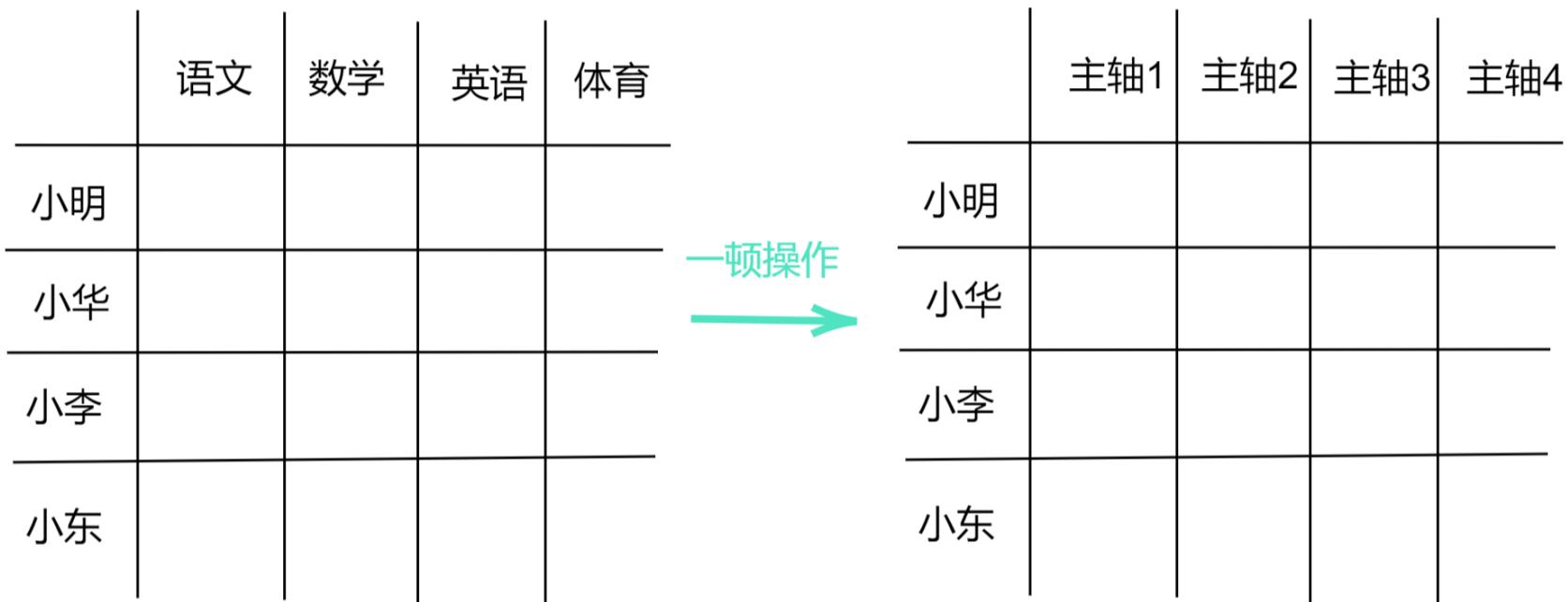
后面还会分析一些东西，这里留个flag

其二，协方差矩阵在这里我们可以给它第一层含义，就是一堆样本的度量矩阵，用于度量某个方向 e 上的方差，

也就是度量这些数据在某个方向上的分散程度的一个矩阵，好比gram矩阵可以度量内积空间任意两个向量的内积一样

第一层含义是我们从推导公式中得来的，但是里面每个元素，也就是每个协方差，到底是什么含义，我们还没讲，这里留个flag

前半文的工作，从一张表得到另一张表



PCA算法（指上图中的一顿操作）

算法核心：

- step1：
计算样本均值 $X.mean(axis=0)$ ，数据作中心化处理 $X=X-X.mean(axis=0)$
设中心化处理后的向量为 $X = [\beta_1, \beta_2, \dots, \beta_p]$
- step2：

$$\text{计算协方差矩阵 } COV_X = \frac{1}{n-1} \begin{bmatrix} \beta_1^T \\ \beta_2^T \\ \vdots \\ \beta_p^T \end{bmatrix} [\beta_1 \ \beta_2 \ \cdots \ \beta_p] = \frac{1}{n-1} X^T X$$

(这里系数为 $\frac{1}{n-1}$ 是使用样本对总体协方差的无偏估计,不用也行, 差不了多少)

- step3：
计算协方差矩阵的特征值，特征向量
由正规矩阵分解定理可知 $COV_X = V \Lambda V^T$, 其中 V 的每一列都是特征向量， Λ 对角元是特征值
我们的算法就是输入 X ,输出 V, Λ, V^T
后续计算 (不放在pca当中)
- step4：
计算新坐标,注意公式中的x,y坐标均为列向量，而表中的x, y均为行向量
- step5：
根据需要，重建，这步没有包含在上面的图中

```
In [5]: def pca(X):
    # 求X中心化后的协方差矩阵
    X = X - X.mean(axis=0)           # 自己验证一下是不是每一列的各个元素减去它所在列的平均值
    COV = torch.matmul(X.T, X) / (X.shape[0] - 1)
```

```
# 求特征值和特征向量
eigenvalue, eigenvector = torch.linalg.eigh(COV)
eigenvalue = torch.where(eigenvalue > 1e-9, eigenvalue, 0)

# 特征值按从大到小排序
index = torch.argsort(-1 * eigenvalue)
eigenvalue = eigenvalue[index]
eigenvector = eigenvector[:, index]

V=eigenvector
Lambda=torch.diag(eigenvalue)

return V,Lambda,V.T
```

In [6]: # test
X=torch.tensor([[1., 2.], [3., 4.]])
pca(X)

Out[6]: (tensor([[0.7071, -0.7071],
 [0.7071, 0.7071]]),
 tensor([[4., 0.],
 [0., 0.]]),
 tensor([[0.7071, 0.7071],
 [-0.7071, 0.7071]]))

对上述结果进行可视化展示

In [7]: X=np.array([[1., 2.], [3., 4.]])
X=pd.DataFrame(X, index=["张三", "李四"], columns=["身高", "体重"])
X

Out[7]: 身高 体重
张三 1.0 2.0
李四 3.0 4.0

In [8]: V, Lambda, Vt=pca(torch.tensor(X.values))
V

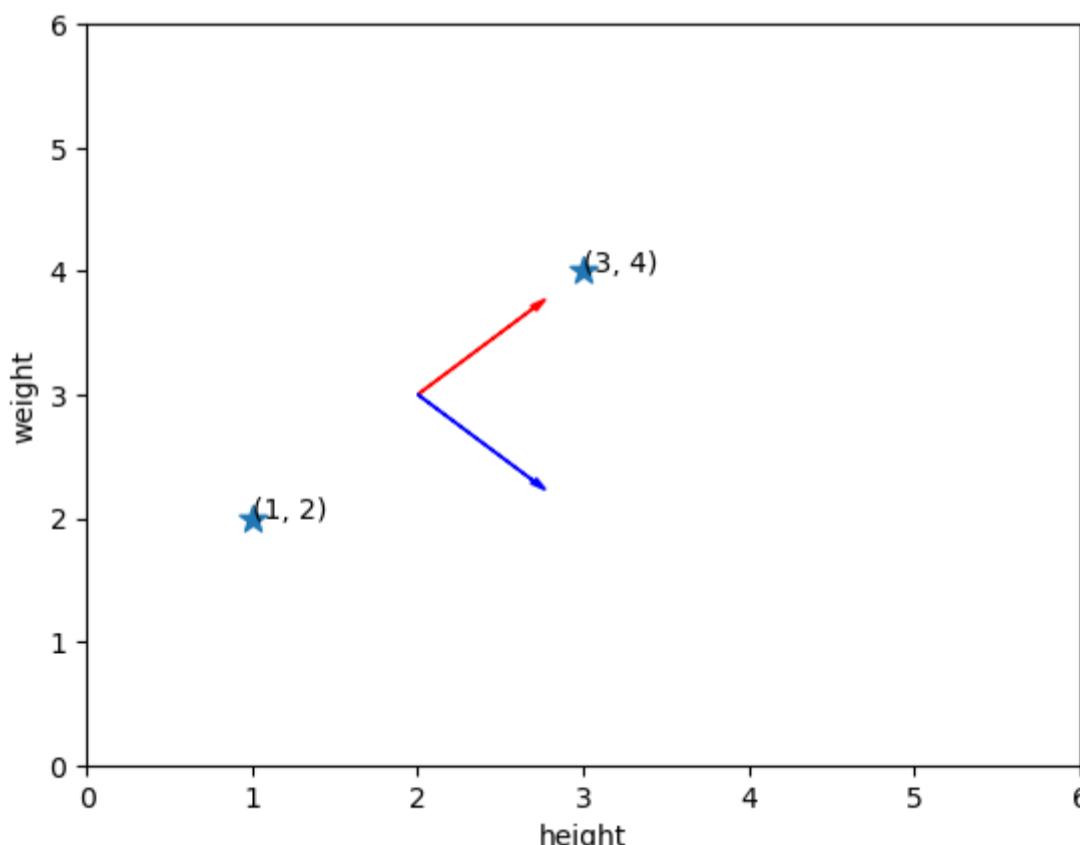
Out[8]: tensor([[0.7071, -0.7071],
 [0.7071, 0.7071]], dtype=torch.float64)

In [9]: height=X["身高"]
weight=X["体重"]
plt.xlabel("height")
plt.ylabel("weight")
plt.xlim(0, 6)
plt.ylim(0, 6)

plt.scatter(height, weight, marker="*", s=100) # 两个人的身高体重坐标
plt.annotate(f"({1}, {2})", (1, 2))
plt.annotate(f"({3}, {4})", (3, 4))

plt.arrow(2, 3, 0.7, 0.7, head_width=0.05, head_length=0.1, color="r") # 第一主成分方向
plt.arrow(2, 3, 0.7, -0.7, head_width=0.05, head_length=0.1, color="b") # 第二主成分方向

Out[9]: <matplotlib.patches.FancyArrow at 0x27555c3bee0>

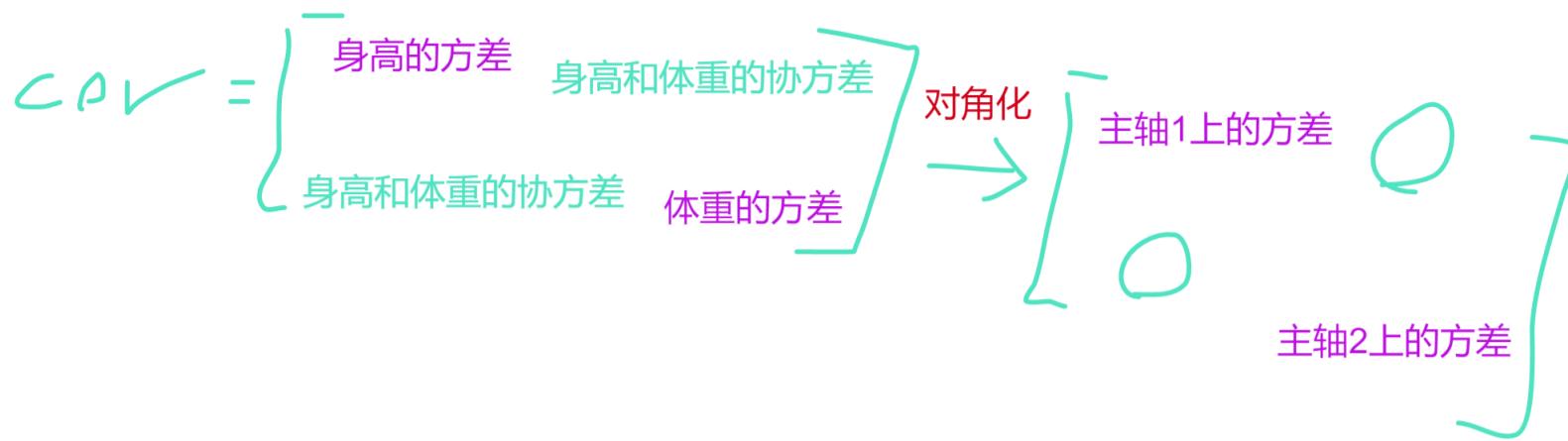


由上图可以看出来：

- 一、当这堆数据的身高和体重强相关时（差不多落在一条直线上），意味着身高和体重有一个是多余的，我们只要知道其一，以及相关系数，就能算出另一个
- 二、当我们使用图中红线和蓝线两条主轴作为新坐标轴时，特征之间的协方差就为0了

三、上述过程实际就是协方差矩阵进行对角化的过程，把除了对角线上的方差之外的所有协方差变成0

四、变换后，主轴上的方差最大，可以提取或存储更多的信息



PCA算法实现的另一种方式

设 $X = X - \bar{X}$

$Cov_X = \frac{1}{n-1} X^T X$, 由正规矩阵分解定理，存在酉矩阵 V 和对角矩阵 Λ , 使得 $Cov_X = \frac{1}{n-1} X^T X = V \Lambda V^T$

其中 V 的列向量是 $\frac{1}{n-1} X^T X$ 的特征向量， Λ 的对角元是 $\frac{1}{n-1} X^T X$ 的特征值

设非零特征值分别为 $\lambda_1, \lambda_2, \dots, \lambda_r$, 对应特征向量分别为 v_1, v_2, \dots, v_r , 那么有 $Cov_X = \lambda_1 v_1 v_1^T + \lambda_2 v_2 v_2^T + \dots + \lambda_r v_r v_r^T$

虽然我们前面的pca算法返回了前n个特征向量，实际上从上式可以看出，返回前r个特征向量就够了，前r个即可无损重建协方差矩阵，但是由于后面我们需要整个V来计算y坐标（也就是表2的全部数值，不降维），所以我们还是返回全部特征向量

对 X 作奇异值分解， $X = U \Sigma V^T$, 其中 Σ 的对角元是 $X^T X$ 的特征值的平方根，即 $\Sigma^2 = \Lambda$, V 就是 $X^T X$ 的特征向量

同样，容易得出 $X = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \dots + \sigma_r u_r v_r^T$

我之前实现的svd算法，因为只用到前r个特征向量，所以只返回这么多，但是内置的svd返回的 V 是全部的特征向量

我们把 X 丢进 svd 当中，可以获得 U, Σ, V , 利用 $\Sigma^2 = \Lambda$, V 就是 $X^T X$ 的特征向量，我们可以间接求出 pca 中的 Λ, V

但是 pca 中的 V 的列向量是 $\frac{1}{n-1} X^T X$ 的特征向量， Λ 的对角元是 $\frac{1}{n-1} X^T X$ 的特征值

而 svd 中的 V 是 $X^T X$ 的特征向量， Σ 的对角元是 $X^T X$ 的特征值的平方根

这样使用 svd 的 V, Σ 来计算 pca 的 V, Λ , 转换起来实属麻烦

所以我们不是直接对 X 进行奇异值分解，而是对 $X / (\sqrt{n-1})$ 进行奇异值分解

结合我们一开始设的 $X = X - \bar{X}$, 最终我们对 $X = \frac{X - \bar{X}}{\sqrt{n-1}}$ 进行奇异值分解

同样，我们想用样本代替总体，那么就使用 $\frac{X - \bar{X}}{\sqrt{n-1}}$ 的无偏估计 $X = \frac{X - \bar{X}}{std(X)}$ 来进行奇异值分解

为了对比计算结果，我们还是对 $X = \frac{X - \bar{X}}{\sqrt{n-1}}$ 进行奇异值分解，别的算法你知道就行了，使用库中的pca就是最后那个，不是我们实现的这个

实际上使用奇异值分解，就是绕了一大圈来求特征值特征向量

```
In [10]: def pca_2(X):
    # normalize the features
    X = (X - X.mean(axis=0)) / torch.sqrt(torch.tensor(X.shape[0]-1))
    U, S, V = torch.linalg.svd(X)

    return V.T, S**2, V
```

```
In [11]: # test
X=torch.tensor([[1., 2.], [3., 4.]])
print(pca_2(X))
print("由此看出，和我们前面的结果一样。注：求解出来的很小的那个数就是0，查一下torch的浮点数0的表示区间")
(tensor([[ 0.7071, -0.7071],
        [ 0.7071,  0.7071]]), tensor([4.0000e+00, 1.8200e-16]), tensor([[ 0.7071,  0.7071],
        [-0.7071,  0.7071]]))
```

由此看出，和我们前面的结果一样。注：求解出来的很小的那个数就是0，查一下torch的浮点数0的表示区间

后半文的工作：

前半文我们核心是围绕这堆数据的信息量（方差）来入手，算是把怎么压缩数据的思路讲清楚了。但是数据降维后需要分析什么没有讲很多，后面我们主要就是对两张表（原始数据和降维后的数据）作一些统计上的分析，通过计算一些统计量，分析数据之间的某种关系，并且给出比较直观的解释

协方差矩阵中的每个协方差是用来刻画什么的？

对于度量空间，我们有比较直观认识的就是欧式距离和夹角，如果我告诉你两个向量之间的欧式距离或者夹角，你大概能对这两个向量之间的关系有个直观的感觉。在统计中，也存在各种度量，比如概率（就是一种测度），用来度量事件（集合）的大小，再比如期望，度量了随机变量（可测映射）

的平均大小（可测函数的积分）。对于协方差矩阵，其对角元素是方差，我们对方差最直观的认识是度量这堆点的离散程度。而非对角元中的元素是两个

特征之间的协方差，那么协方差这个数值到底刻画了这两个特征之间的什么内在联系呢，由于讲起来特别麻烦，还要画图，这里就不写了，

不过这里有一份不错的参考，打开这个知乎回答里面提到的3个链接，里面对协方差有一个详细的解释。reference

简单来说就是我给你一个相关系数，和这堆点的中心（平均），你能根据相关系数大致判断大多数点位于哪几个象限，大致呈现一个什么分布，什么形状，胖的还是瘦的

比如一堆人的身高和体重，大致就呈现一种线性关系，高的重，矮的轻，但也不是很相关，所以这种线性不会很细，会胖

话不多说，这里分析以下3种花的花萼长度和花瓣长度有什么关系

```
In [12]: setosa_x=setosa.loc[:, "SepalLengthCm"]          # 拿setosa的花萼长度作为横坐标
setosa_y=setosa.loc[:, "PetalLengthCm"]            # 拿setosa的花瓣长度作为纵坐标
cov_setosa=(setosa_x-setosa_x.mean()).T.dot(setosa_y-setosa_y.mean())

versicolor_x=versicolor.loc[:, "SepalLengthCm"]
versicolor_y=versicolor.loc[:, "PetalLengthCm"]
cov_versicolor=(versicolor_x-versicolor_x.mean()).T.dot(versicolor_y-versicolor_y.mean())

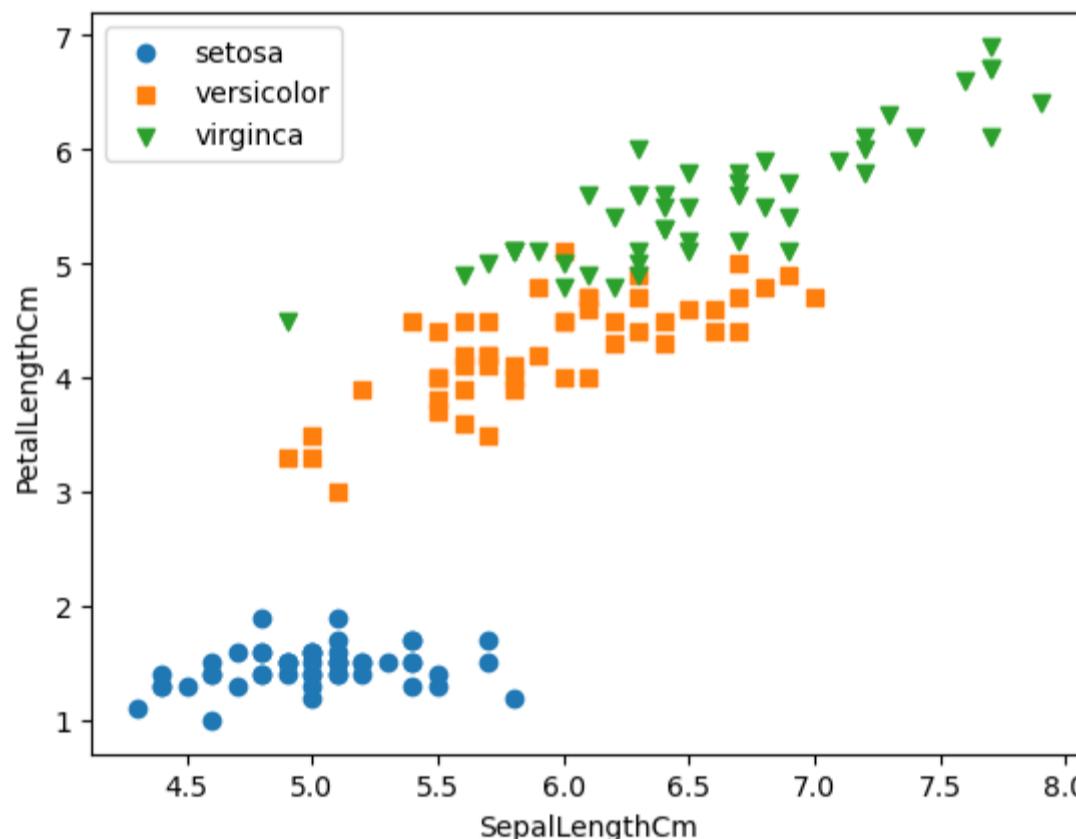
virginca_x=virginca.loc[:, "SepalLengthCm"]
virginca_y=virginca.loc[:, "PetalLengthCm"]
cov_virginca=(virginca_x-virginca_x.mean()).T.dot(virginca_y-virginca_y.mean())

cov_setosa, cov_versicolor, cov_virginca
```

```
Out[12]: (0.7907999999999998, 8.962000000000002, 14.861200000000004)
```

```
In [13]: plt.scatter(setosa_x, setosa_y, marker="o", label="setosa")      # setosa的图形使用圆形表示
plt.scatter(versicolor_x, versicolor_y, marker="s", label="versicolor")
plt.scatter(virginca_x, virginca_y, marker="v", label="virginca")
plt.xlabel("SepalLengthCm")
plt.ylabel("PetalLengthCm")
plt.legend()                      # 显示设置的label图例
```

```
Out[13]: <matplotlib.legend.Legend at 0x27555ac2ee0>
```



setosa的花瓣长度与花萼长度之间的协方差为0.79，也就是以数据中心为原点看，散落在四个象限的点与原点构成的有符号矩阵面积累加起来是0.79，

说明1, 3象限的正矩形面积比2, 4象限的负矩形面积多了0.79，所以四个象限的点差不多，没有什么相关性，图上蓝色点就是这种特点

versicolor的花瓣长度与花萼长度之间的协方差为8.96，对应图中橘色点，有点相关性了，横坐标花萼如果从5变成6了，那么花瓣大概率也会变大，这就是正相关

第三种花，相关性更强了，看图上绿色倒三角，一目了然

鸢尾花PCA分析

开始分析之前，先把两张表弄出来

table_X就是原来的数据

table_Y就是新坐标系下的坐标 $y = A_p^T(x - \bar{x})$

```
In [14]: table_X=data.values
table_X[:5, :]
```

```
Out[14]: array([[5.1, 3.5, 1.4, 0.2],
 [4.9, 3., 1.4, 0.2],
 [4.7, 3.2, 1.3, 0.2],
 [4.6, 3.1, 1.5, 0.2],
 [5., 3.6, 1.4, 0.2]])
```

取全部的主轴p=4

```
In [15]: x=torch.tensor(table_X)
V,Lambda,Vt=pca(x)
table_Y=Vt.matmul(torch.tensor(table_X-table_X.mean(axis=0)).T).T
table_Y[:5,:]
```

```
Out[15]: tensor([[ 2.6842e+00,  3.2661e-01,  2.1512e-02,  1.0062e-03],
                  [ 2.7154e+00, -1.6956e-01,  2.0352e-01,  9.9602e-02],
                  [ 2.8898e+00, -1.3735e-01, -2.4709e-02,  1.9305e-02],
                  [ 2.7464e+00, -3.1112e-01, -3.7672e-02, -7.5955e-02],
                  [ 2.7286e+00,  3.3392e-01, -9.6230e-02, -6.3129e-02]],  
dtype=torch.float64)
```

取两个主轴p=2和上面的结果对比，发现完全一样，所以求出全部主轴坐标后再删掉多余的也行

```
In [16]: table_Y2=Vt[:2,:].matmul(torch.tensor(table_X).T).T
table_Y2[:5,:]
```

```
Out[16]: tensor([[-2.8271,  5.6413],
                  [-2.7960,  5.1452],
                  [-2.6215,  5.1774],
                  [-2.7649,  5.0036],
                  [-2.7828,  5.6486]],  
dtype=torch.float64)
```

到底取多少条主轴比较合适——碎石图

我们知道特征值代表了这堆数据在各个轴上的能量，所以我们通过计算前面主要部分的能量占全部总能量的百分比，就可以决定选用多少主轴了

$$\frac{\lambda_1 + \lambda_2 + \dots + \lambda_t}{\lambda_1 + \lambda_2 + \dots + \lambda_p} > q$$

$q = 75\%, 80\%, 85\%, 90\%, \text{ or } 95\%$

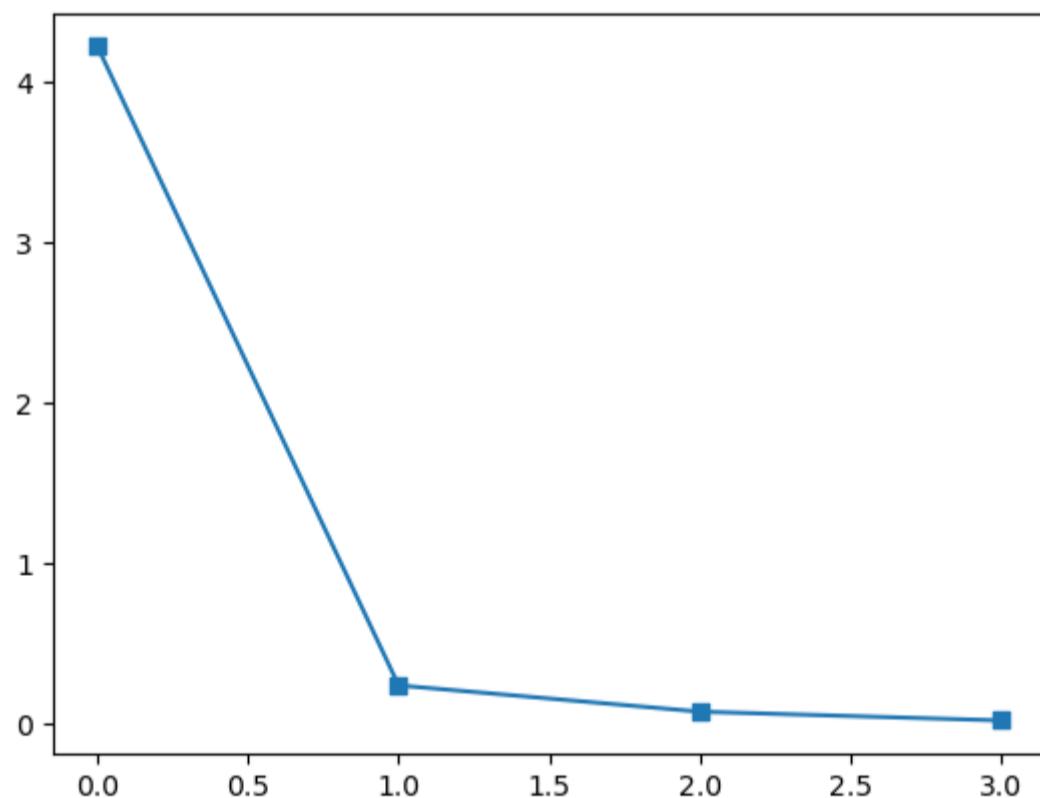
主轴少的时候，可以慢慢算，主轴太多的时候，可以画出碎石图，观察能量转折点，在转折点右边来一刀就是我选的分界点
碎石图(Scree plot)很简单，下面直接看图

```
In [17]: Lambda=Lambda.diag()
Lambda
```

```
Out[17]: tensor([4.2248, 0.2422, 0.0785, 0.0237],  
dtype=torch.float64)
```

```
In [18]: plt.scatter(np.arange(4),Lambda.numpy(),marker="s")
plt.plot(np.arange(4),Lambda.numpy())
```

```
Out[18]: [<matplotlib.lines.Line2D at 0x27555e34340>]
```



从图上可以看出，第二个点处急转直下，分别取1, 2, 3, 4个主轴，能量占比分别为

$$\begin{aligned}\lambda_1 &= 4.22824170603486, \lambda_2 = 0.242670747928634 \\ \lambda_3 &= 0.0782095000429194, \lambda_4 = 0.0238350929734494 \\ \frac{\lambda_1}{\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4} &= 0.924618723201727 \\ \frac{\lambda_1 + \lambda_2}{\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4} &= 0.977685206318795 \\ \frac{\lambda_1 + \lambda_2 + \lambda_3}{\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4} &= 0.994787816126725 \\ \frac{\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4}{\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4} &= 1.000000000000000\end{aligned}$$

注：上面的结果取自李政轩的统计机器学习课程，所以数值跟我上面算的差了那么一点点，可能不同软件精度不同
我们就取前两个主轴，下面进行可视化展示

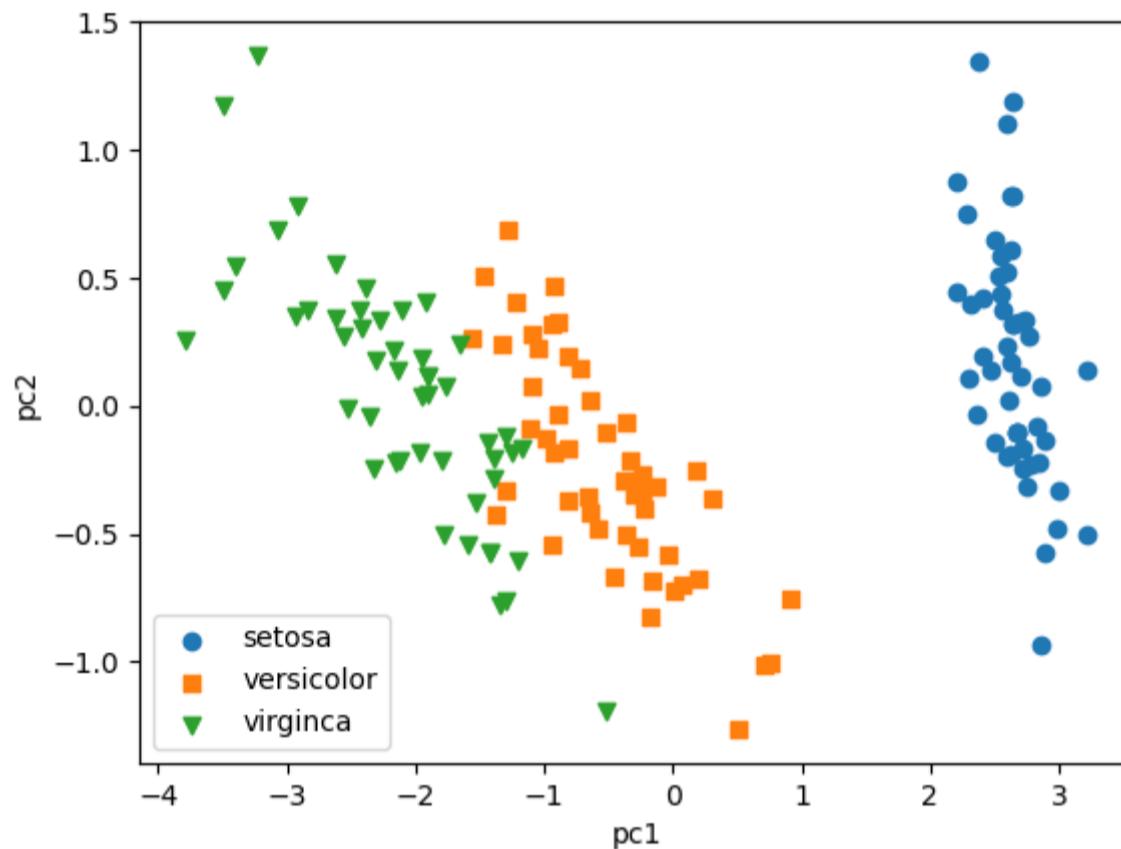
```
In [19]: setosa_x=table_Y[:50, 0].numpy()      # 拿setosa的萼片长度作为横坐标
setosa_y=table_Y[:50, 1].numpy()      # 拿setosa的花瓣长度作为纵坐标

versicolor_x=table_Y[50:100, 0].numpy()
versicolor_y=table_Y[50:100, 1].numpy()

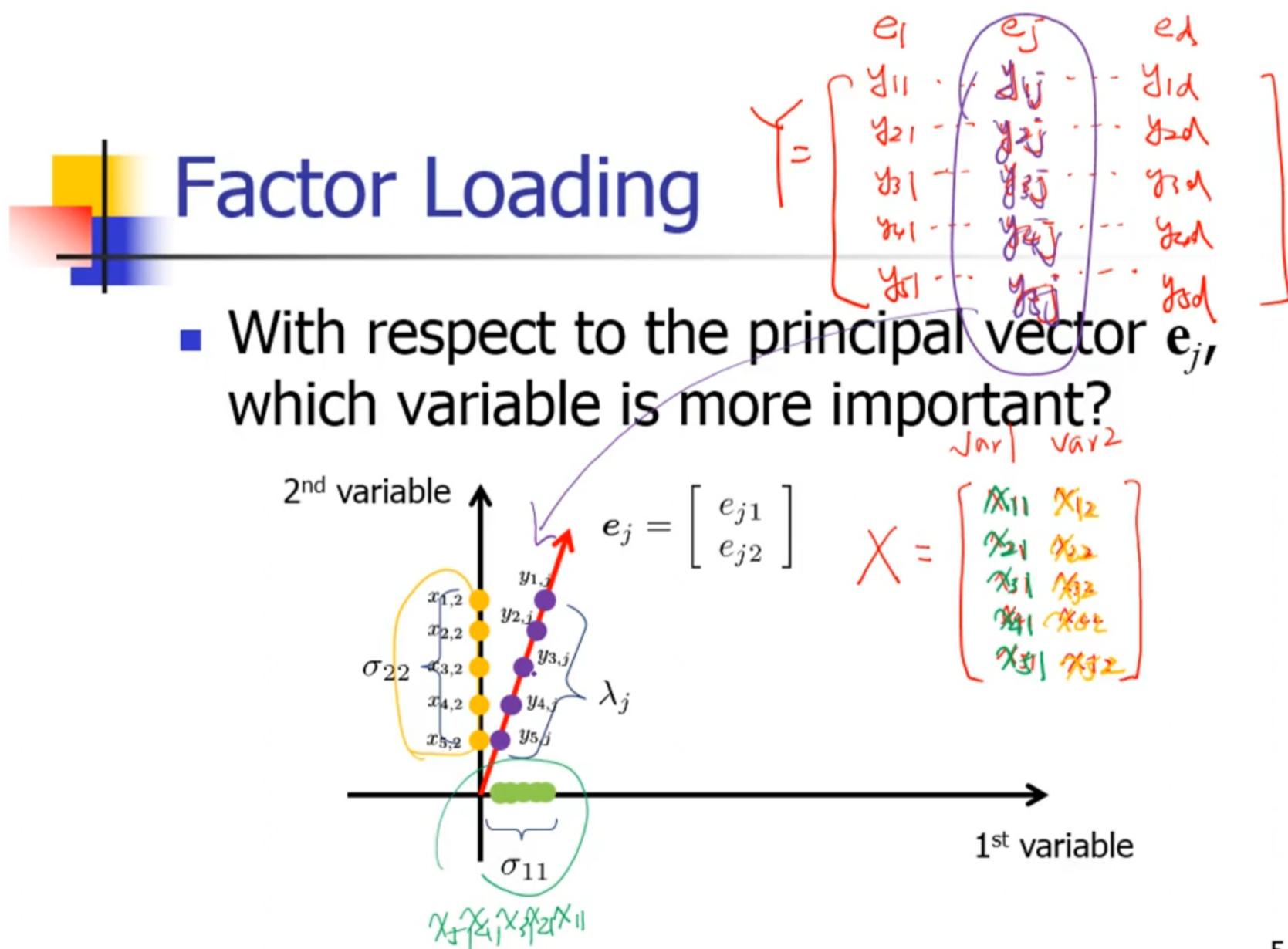
virginica_x=table_Y[100:150, 0].numpy()
virginica_y=table_Y[100:150, 1].numpy()

plt.scatter(setosa_x, setosa_y, marker="o", label="setosa")      # setosa的图形使用圆形表示
plt.scatter(versicolor_x, versicolor_y, marker="s", label="versicolor")
plt.scatter(virginica_x, virginica_y, marker="v", label="virginica")
plt.xlabel("pc1")
plt.ylabel("pc2")
plt.legend()
```

Out[19]: <matplotlib.legend.Legend at 0x27555deb220>



旧特征（花瓣长、宽，萼片长、宽）与新特征（4个主轴：pc1,pc2,pc3,pc4）有什么关系呢？



看上图，我们取5个样本，分别投影到原先坐标系的前两个坐标轴上（前两个特征轴）和第j个主轴上，就变成了上面这幅图的样子。我们不难看出，数据在第二个变量上的分散程度和主轴上的分散程度比较接近，说明第2个变量与第j个主轴更相关，第j个主轴上的能量主要来自第

二个变量

既然第j个主轴的信息主要来自第二个变量，比如第二个变量叫体重，那么我们可以近似认为第j个主轴近似就是体重，它和第一个变量（月收入）没多少关系

统计上的数值表现为，第二个变量方差很大的时候，主轴上的方差也很大，如果采集的数据包含第一个变量，第二个变量，第j个主轴这3个特征数据，那么

我们可以随便丢掉第二个变量或第j个主轴，因为这两个东西可以看成一个东西，信息重复了

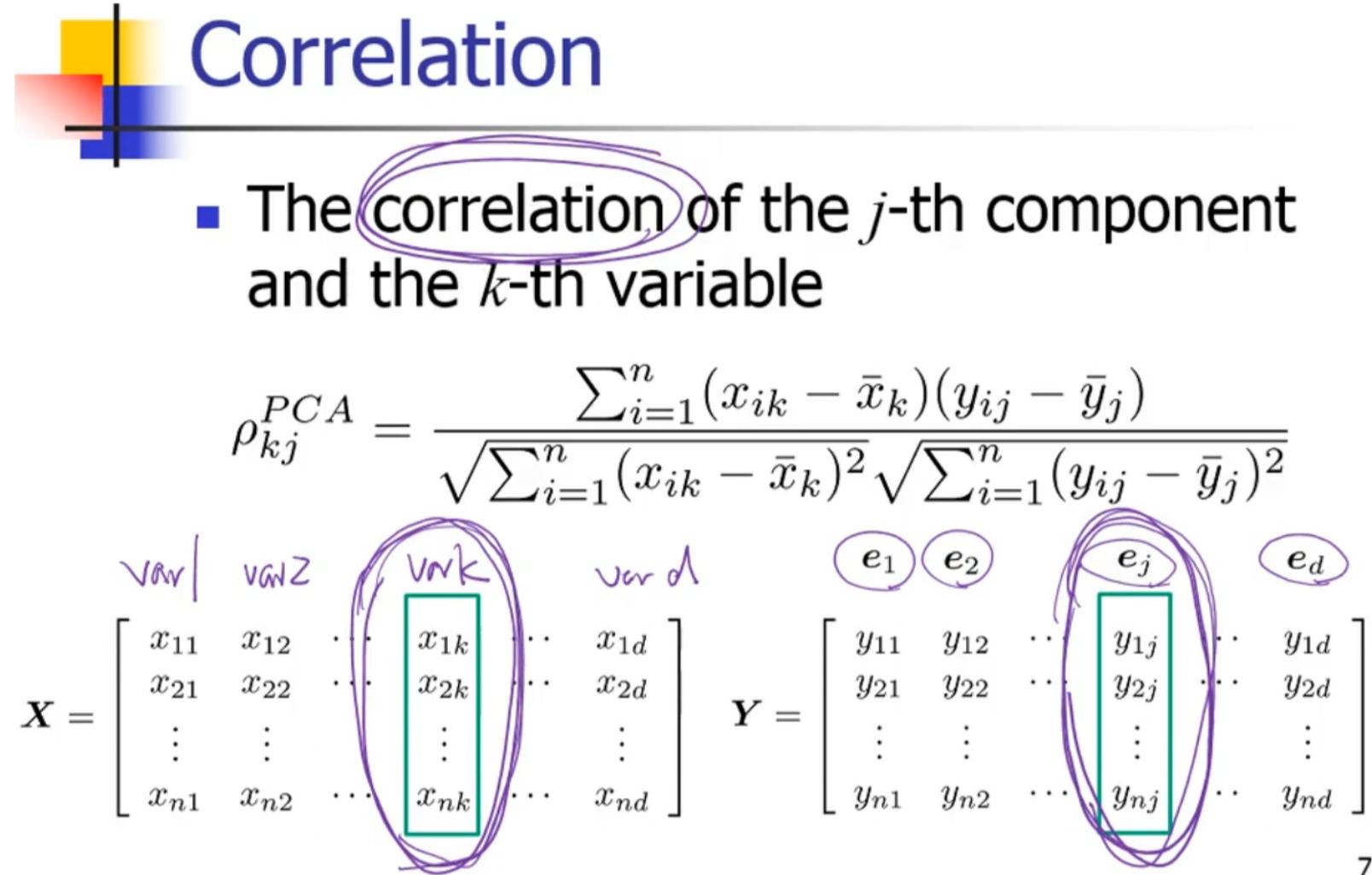
这也是为什么pca后，原始数据就可以不看了，因为信息已经被我提取到主轴上面了

因子分析

如下图，就是计算第k个特征和第j个主轴之间的相关系数，特征和主轴其实都是对样本的某个特征，

这里计算相关系数和前面讲的计算协方差，度量的都是两个特征之间的相关关系

公式就是数据归一化后的内积



7

可视化展示Biplot

第一步：以pc1, pc2为坐标轴，画出所有样本的点，下图中的红点

第二步：分别计算第1, 2, 3, 4个变量与pc1, pc2之间的相关系数，总共8个数，放在 e_1, e_2 里面。这个地方也叫贡献率，知道就行

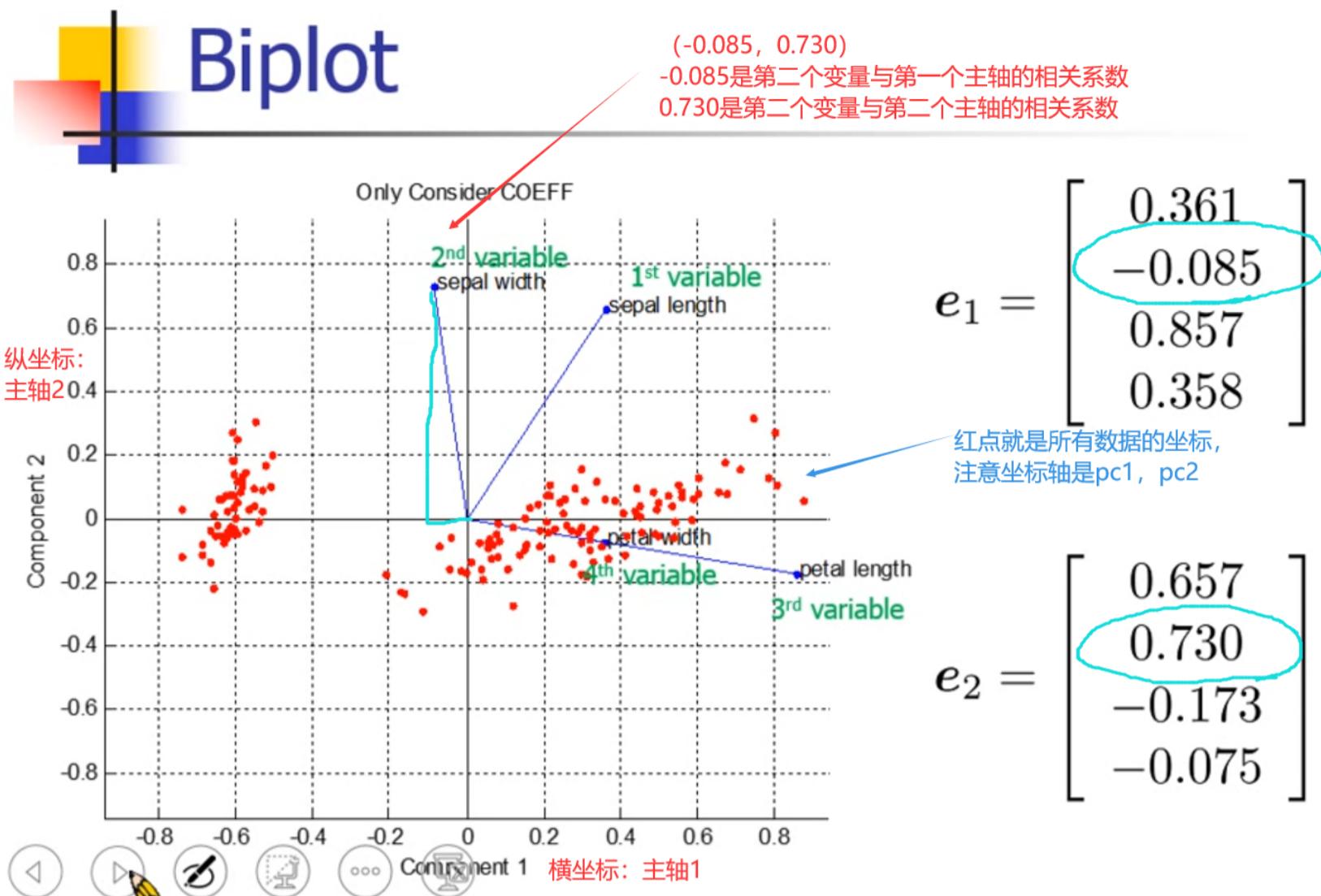
第三步：把与pc1的相关系数(e_1 的每个数值)当成横坐标x，把与pc2的相关系数(e_2 的每个数值)当成纵坐标，画在图上，用来表示4个变量轴

第四步：跟分析上面的那张图片一样分析下面这张图，可以看出，第1, 3, 4变量与pc1更相关，

直观上挨得近，数值上说明三个变量的能量都在pc1上，不信你算算三个变量的方差总和是不是和pc1上的方差很接近

第2个变量与pc2更为相关

由于时间问题，这里不详细计算了，有兴趣的可以自己算算



数据重建

重建公式为：

$$\hat{x} = \bar{x} + A_t \hat{y}$$

我们不妨就拿两个主轴进行重建

注意计算的时候，表里面的向量是横着摆放的，而公式里面的向量是列向量，竖着的

```
In [20]: x=torch.tensor(table_X)
V,Lambda,V_T=pca(x)
table_Y=V_T.matmul(torch.tensor(table_X).T).T
x_rebuild=x.mean(axis=0)+V[:, :2].matmul(table_Y[:, :2].T).T
x_rebuild[:5, :]
```

```
Out[20]: tensor([[10.5694,  6.9380,  5.1888,  1.7917],
 [10.2323,  6.5785,  5.2492,  1.8176],
 [10.1904,  6.6163,  5.0942,  1.7526],
 [10.1282,  6.4777,  5.2475,  1.8170],
 [10.5581,  6.9470,  5.1494,  1.7752]], dtype=torch.float64)
```

使用机器学习库快速计算

先给出前面自己实现的算法的结果

表2的前5条数据，降维成2维

```
In [21]: x=torch.tensor(table_X)
V,Lambda,Vt=pca(x)
table_Y=Vt.matmul(torch.tensor(table_X-table_X.mean(axis=0)).T).T
table_Y[:5, :2]
```

```
Out[21]: tensor([[ 2.6842,   0.3266],
 [ 2.7154,  -0.1696],
 [ 2.8898,  -0.1373],
 [ 2.7464,  -0.3111],
 [ 2.7286,   0.3339]], dtype=torch.float64)
```

数据重建

```
In [22]: x_rebuild=x.mean(axis=0)+V[:, :2].matmul(table_Y[:, :2].T).T
x_rebuild[:5, :]
```

```
Out[22]: tensor([[ 5.0872,  3.5132,  1.4020,  0.2111],
 [ 4.7502,  3.1537,  1.4625,  0.2369],
 [ 4.7082,  3.1915,  1.3075,  0.1719],
 [ 4.6460,  3.0529,  1.4608,  0.2364],
 [ 5.0759,  3.5221,  1.3627,  0.1946]], dtype=torch.float64)
```

再给出使用sklearn的包的结果

```
In [23]: from sklearn.decomposition import PCA
pp=PCA(n_components=2)
pp.fit(table_X)
```

表2的前5条数据，降维成2维

```
In [24]: # 取2维，输出新表的前5条数据，并与前面自己算法输出的结果进行对比
table_Y=pp.transform(table_X)
table_Y[:5, :]
```

```
Out[24]: array([[-2.68420713,  0.32660731],
 [-2.71539062, -0.16955685],
 [-2.88981954, -0.13734561],
 [-2.7464372 , -0.31112432],
 [-2.72859298,  0.33392456]])
```

数据重建

```
In [25]: x_rebuild=pp.inverse_transform(table_Y)
x_rebuild[:5, :]
```

```
Out[25]: array([[5.08718247, 3.51315614, 1.4020428 , 0.21105556],
 [4.75015528, 3.15366444, 1.46254138, 0.23693223],
 [4.70823155, 3.19151946, 1.30746874, 0.17193308],
 [4.64598447, 3.05291508, 1.46083069, 0.23636736],
 [5.07593707, 3.5221472 , 1.36273698, 0.19458132]])
```

sklearn中的对应属性

sklearn中的pca用奇异值分解实现的，里面有复杂的对应转换关系，有兴趣的可以看看源代码
需要注意的是每个实现的细节，和我们实现的多多少少有细微差异，一不注意就会搞错

[doc](#)
[source code](#)

数据的均值

```
In [26]: pp.mean_, x.mean(axis=0)
```

```
Out[26]: (array([5.84333333, 3.054      , 3.75866667, 1.19866667]),
 tensor([5.8433, 3.0540, 3.7587, 1.1987], dtype=torch.float64))
```

特征值（主轴上的方差，能量，解释量，贡献量）

```
In [27]: pp.explained_variance_, Lambda.diag()
```

```
Out[27]: (array([4.22484077, 0.24224357]),
 tensor([4.2248, 0.2422, 0.0785, 0.0237], dtype=torch.float64))
```

特征向量的转置

```
In [28]: pp.components_, V.T
```

```
Out[28]: (array([[ 0.36158968, -0.08226889,  0.85657211,  0.35884393],
 [ 0.65653988,  0.72971237, -0.1757674 , -0.07470647]]),
 tensor([[-0.3616,  0.0823, -0.8566, -0.3588],
 [ 0.6565,  0.7297, -0.1758, -0.0747],
 [ 0.5810, -0.5964, -0.0725, -0.5491],
 [ 0.3173, -0.3241, -0.4797,  0.7511]], dtype=torch.float64))
```

理论上知道上面3个东西，就什么都能做了，比如输出表2

其他的就不重复了，需要注意的是每个矩阵的行列是怎么摆放的，搞清楚表怎么对应公式

```
In [29]: table_Y=(table_X-table_X.mean(axis=0)).dot(pp.components_.T)
table_Y[:5, :]
```

```
Out[29]: array([[-2.68420713,  0.32660731],
 [-2.71539062, -0.16955685],
 [-2.88981954, -0.13734561],
 [-2.7464372 , -0.31112432],
 [-2.72859298,  0.33392456]])
```

其他库的PCA接口：

matlab: [主轴 (特征向量) , 表2, 特征值]=[coeff,score,latent]=pca(x)

sklearn的包使用步骤小结：

- 1.实例化模型：model=PCA(n_components=2) # 可以指定n_components属性，也可以不指定
- 2.模型训练：model.fit()
- 3.使用方法或者属性获取重要结果，常用到的重要结果获取方式如下：
Y=model.transform()
new_X=model.inversetransform()

```
components=model.components
eigenvalues=model.explainedvariance
```

手写数字PCA降维分析

由于PCA基本分析的东西就是上面那些，这里不重复上面的内容，讲点别的东西

数据准备

```
In [30]: with open("data/mnist_idx3-ubyte", "rb") as f:
    # 指针移动到第16个字节处
    f.seek(16)
    # 把后面所有的字节填入uint8中，返回一维数组array
    # 把这个一维数组reshape成60000x784的形状，每一行就是一张图片
    images = np.fromfile(f, dtype="uint8").reshape(-1, 784)

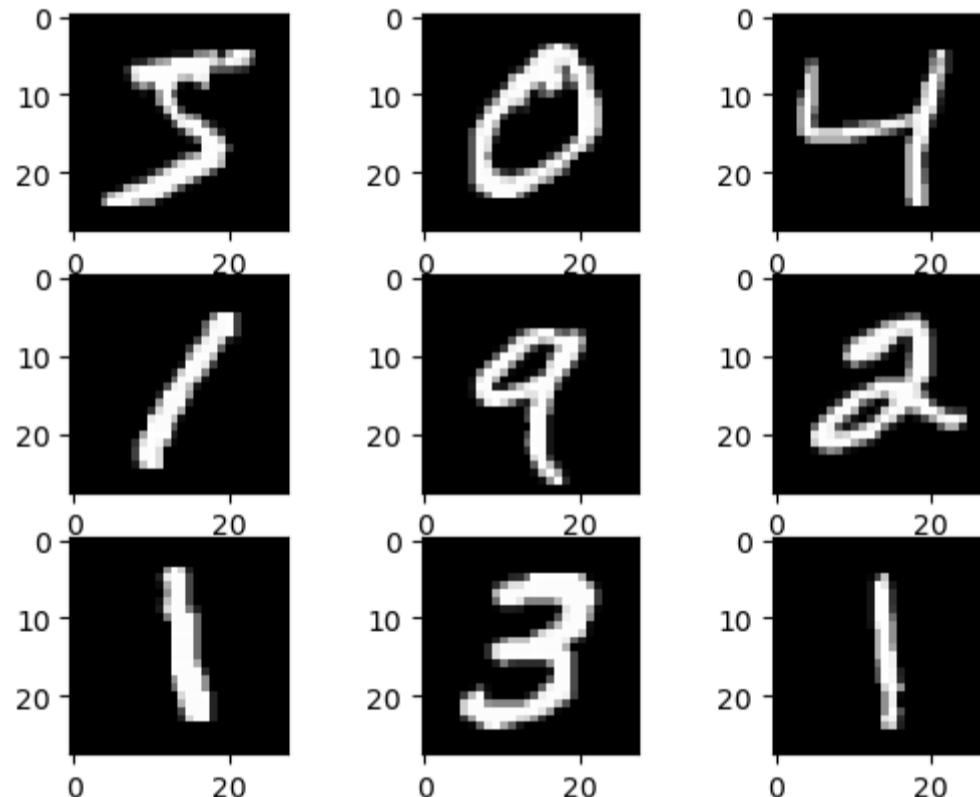
with open("data/mnist-labels_idx1-ubyte", "rb") as f:
    f.seek(8)
    labels = np.fromfile(f, dtype="uint8")

images.shape, labels.shape
```

Out[30]: ((60000, 784), (60000,))

```
In [31]: fig, axes=plt.subplots(3, 3)
for i in range(3):
    for j in range(3):
        axes[i][j].imshow(images[i*3+j].reshape(28, 28), cmap='gray')
labels[:9]
```

Out[31]: array([5, 0, 4, 1, 9, 2, 1, 3, 1], dtype=uint8)



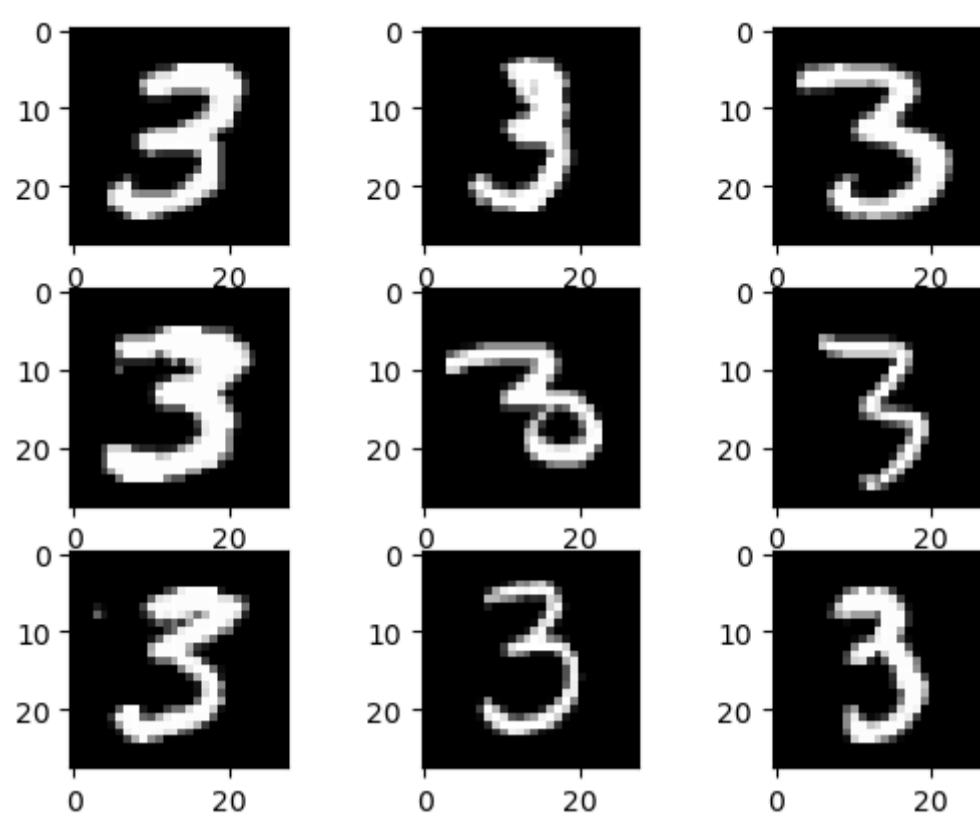
```
In [32]: # 取出前8000张图片的所有数字3拿来做PCA分析
images_=images_[:8000, :]
labels_=labels_[:8000]
index=(labels_==np.array(3))
images_.shape, labels_.shape, index.sum()
```

Out[32]: ((8000, 784), (8000,), 811)

```
In [33]: images_3=images_[index, :]
images_3.shape
```

Out[33]: (811, 784)

```
In [34]: fig, axes=plt.subplots(3, 3)
for i in range(3):
    for j in range(3):
        axes[i][j].imshow(images_3[i*3+j].reshape(28, 28), cmap='gray')
```

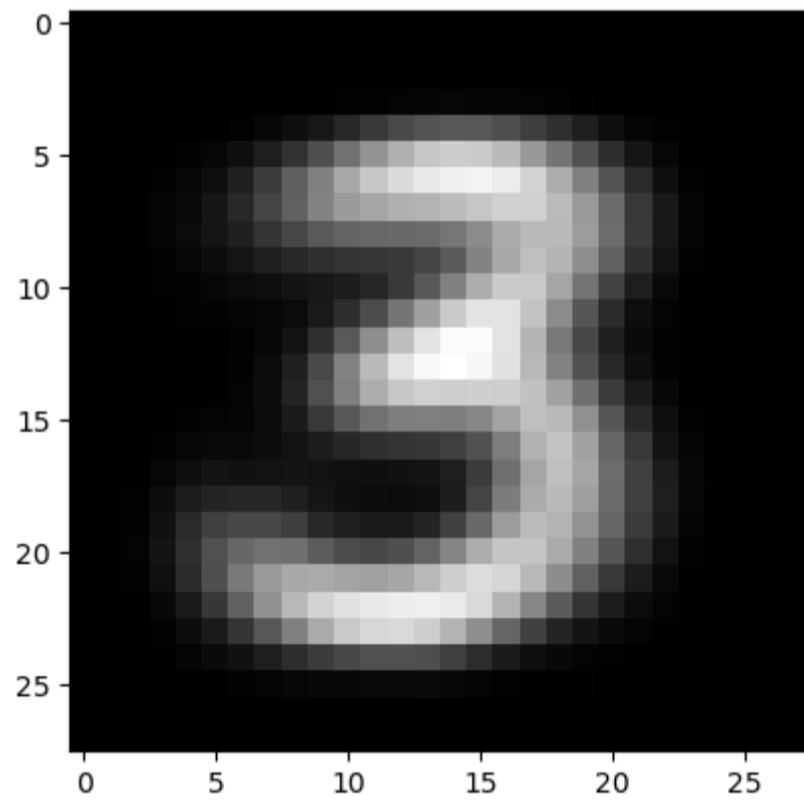


PCA降维

```
In [35]: table_X=images_3
table_X=torch.tensor(table_X,dtype=torch.float)
V,Lambda,Vt=pca(table_X)
table_Y=Vt.matmul((table_X-table_X.mean(axis=0)).T).T
```

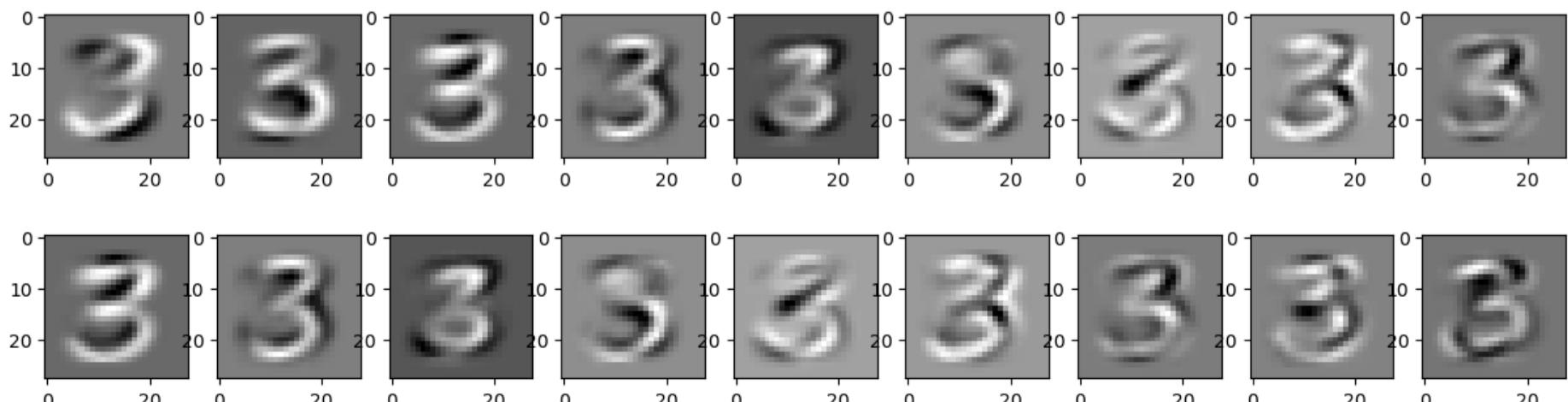
一、画出均值向量

```
In [36]: plt.imshow(table_X.mean(axis=0).reshape(28, 28), cmap="gray")
Out[36]: <matplotlib.image.AxesImage at 0x2755d4de0d0>
```



二、画出前18个主轴向量

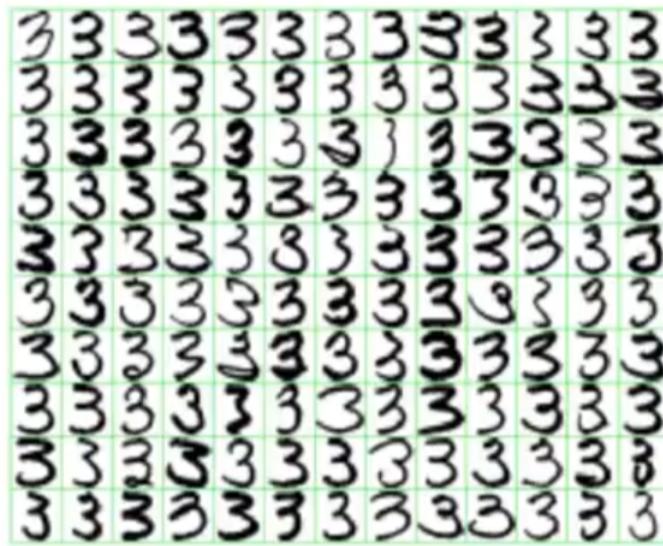
```
In [37]: fig,axes=plt.subplots(2,9)
fig.set_size_inches(15, 4)
for i in range(2):
    for j in range(9):
        axes[i][j].imshow(Vt[i*2+j,:].reshape(28, 28), cmap='gray')
```



三、回顾重建公式

$$\begin{aligned}\hat{\mathbf{x}} - \bar{\mathbf{x}} &\approx y_1 \mathbf{e}_1 + y_2 \mathbf{e}_2 + \cdots + y_t \mathbf{e}_t = \mathbf{A}_p \mathbf{y} \\ \hat{\mathbf{x}} &\approx \bar{\mathbf{x}} + \mathbf{A}_p \mathbf{y} \\ &\approx \bar{\mathbf{x}} + y_1 \mathbf{e}_1 + y_2 \mathbf{e}_2 + \cdots + y_t \mathbf{e}_t\end{aligned}$$

即某一张数字3=平均3+各个主轴的加权求和 (权重为这个3在每个主轴上的得分, 即表2中的坐标)

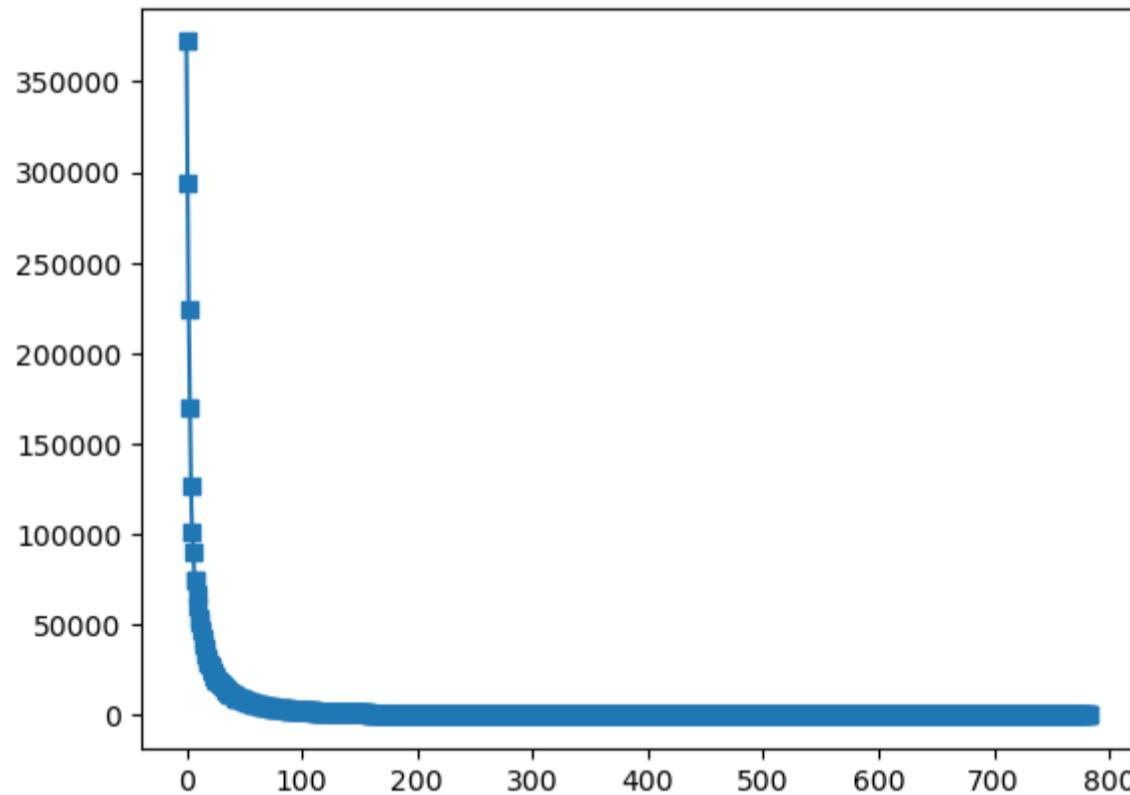


$$\text{3} = \text{3} + y_1 \cdot \text{3} + y_2 \cdot \text{3}.$$

四、由碎石图确定数字3的真正维度

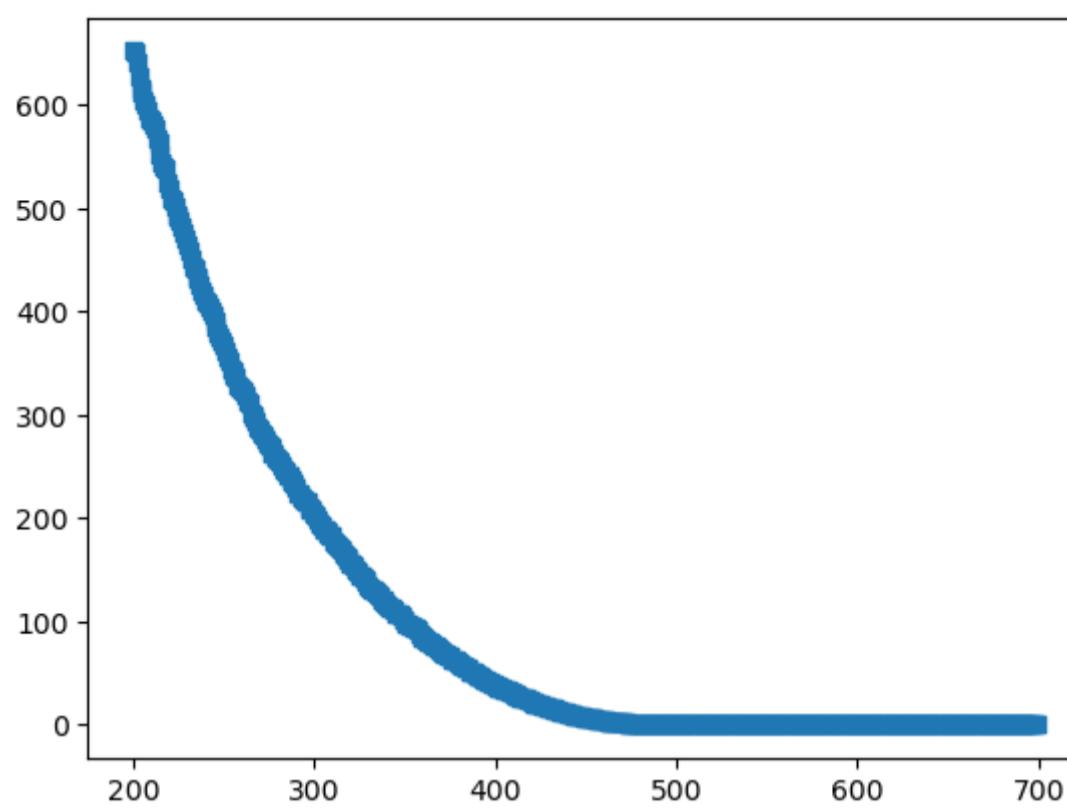
```
In [38]: Lambda=Lambda.diag().numpy()
plt.scatter(np.arange(784),Lambda,marker="s")
plt.plot(np.arange(784),Lambda)

Out[38]: [<matplotlib.lines.Line2D at 0x2755d8188b0>]
```



```
In [39]: plt.scatter(np.arange(200, 700),Lambda[200:700],marker="s")
plt.plot(np.arange(200, 700),Lambda[200:700])

Out[39]: [<matplotlib.lines.Line2D at 0x2755d9df430>]
```



In [40]: `Lambda[:450].sum() / Lambda.sum()`

Out[40]: 0.99996173

问题：

为什么主轴具有数字3的形态？

我们不妨假定数字3位于784维高维空间中的某个低维子空间中，比如450维（包含了99%以上的信息了，够了）

但是450维图片可以是五花八门，随便一张450维图片是数字3的可能性几乎为0.

对于数字3的平均值，因为它是数字3所在空间元素的加法运算，所以还在这个子空间当中，所以平均值有3的样子合情合理

但是主轴并不是子空间元素的简单线性运算而来，而是协方差矩阵的特征向量，从运算上很难得出这个特征向量属于3所在子空间

如果有数学大神，不妨去证明一下到底在不在那个子空间里

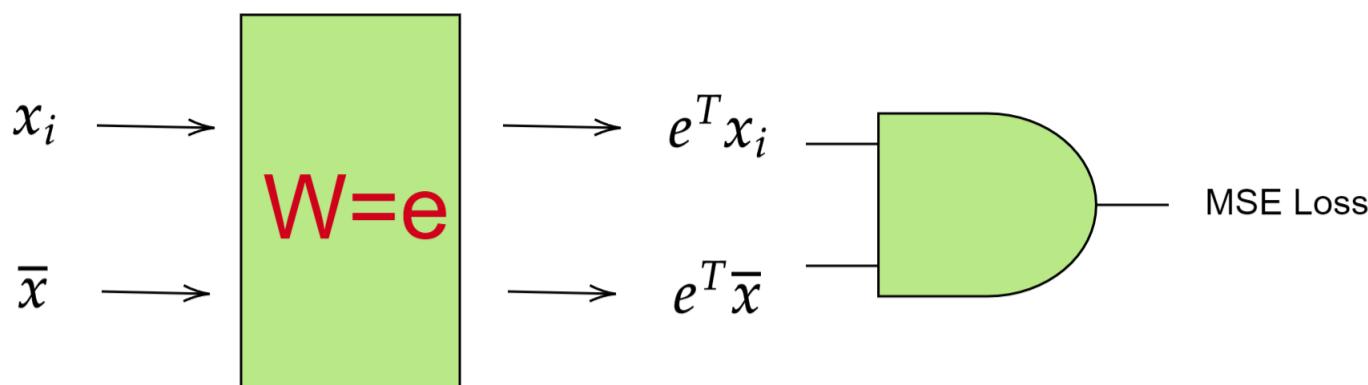
另外一个思路就是，即便不在那个子空间里，能否求出与子空间（或者子空间某个元素，降低计算难度）的距离，如果距离很小，可以认为主轴只是子空间中某个3的扰动

最后一个思路就是从深度学习角度考虑，特征向量就是我们一开始求投影到某个轴上的方差最大这个优化问题的解

不妨令这个轴为网络参数，只设置一层网络，那么就是如下损失函数的优化问题：

$$Loss = \frac{1}{n} \sum_{i=1}^n (e^T(x_i - \bar{x}))^2$$

注意约束项通过正则化近似实现



1. 可以看成一个自监督模型，标签用自己的均值投影到 e 上面
2. 为什么优化后， e 会和样本这么像（相似）
3. word embedding模型中，为什么训练后近义词的编码那么相似
4. Cycle GAN中，为什么优化后， P_G 能找到对应的分布
5. 上面的点乘自监督基本单元能否替换transformer，不能的话适用于什么情况，或者本质就是直接计算mse loss的全连接神经网络？

几个关于模型训练与优化的理解：

- (1). 模型是为了让方差变大，也就是负方差变小，由于是负平方，所以这个loss可以无限变小下去
- (2). 模型通过调节w轴的长度和方向来使方差变大，如果输入x是0-255之间的数值，那么会导致各个维度的梯度差异很大，大梯度的维度会率先爆炸。令x, w都是一维， $loss = (xw)^2$, 那么 $dloss = d(xw)^2 = 2x^2wdw$, 迭代： $w_1 = w_0 + 2x^2w_0 \approx 2x^2w_0$, $w_2 = w_1 + 2x^2w_1 \approx (2x^2)^2w_0$ ，由此可以看出，如果x很大，梯度很大，梯度很大就是导致w模长很大，按x指数级别增长，个别维度很快就会数值溢出，这时候可以让x除以D（维度）或者让学习率变小。
- 但是学习率变小需要调的非常非常小，很难调，建议令x除以D，或者255
- (3). Adam改变了梯度方向，有自己的学习偏好，需要调整正则化系数（这里lambda=0.2~0.5，L2正则化）。而使用SGD不存在此问题，只要通过梯度裁剪（缩放），保证梯度限制在一定范围，从而更新过程中w缓慢增长，只要w一直更新下去不爆炸，一定会收敛。
- (4). 由于SGD只要能正常训练，必定会收敛，所以有几种控制w模长的办法：一、 $x/255$ 或者 wx/D 保证梯度很小；二、梯度裁剪；三、超级超级小的学习率，较大的训练次数
- (5). SGD比Adam更容易学对方向，没有特殊偏好

```
In [41]: import torch
import torch.nn as nn
inputs=torch.tensor(images_3,dtype=torch.float)
inputs=(inputs-inputs.mean(axis=0))
```

方案一：SGD+输入归一化

```
In [42]: inputs2=(inputs-inputs.mean(axis=0))/255

class mymodel(nn.Module):
    def __init__(self):
        super(mymodel, self).__init__()
        self.W=nn.Parameter(torch.randn((784, 1), dtype=torch.float), requires_grad=True)
    def forward(self, x):
        return x.mm(self.W)

model=mymodel()
loss_fn=nn.MSELoss()
learning_rate=0.001
optimizer=torch.optim.SGD(model.parameters(), lr=learning_rate)

num_epoches=600

for epoch in range(num_epoches):
    outputs=model(inputs2)
    r=(torch.sum(torch.square(model.W))-800)**2
    loss=-loss_fn(outputs, torch.zeros(811, 1))
```

```
# 更新参数
optimizer.zero_grad()
loss.backward()
optimizer.step()

# 每隔 200 次训练输出一次总损失
if epoch%200==0 or epoch==num_epochs-1:
    with torch.no_grad():
        print("W模长", torch.sqrt(torch.sum(torch.square(model.W))))
        print("-sigma2", -loss_fn(outputs, torch.zeros(811, 1)))
        print("loss", loss)
        print("\n")
        print('epoch: {}, loss: {}'.format(epoch+1, loss))

print('done!')
for p in model.parameters():
    np.set_printoptions(suppress=True)
    np.set_printoptions(precision=3)
    imgg=np.copy(p.detach().numpy())

    # print(np.square(imgg).T)
    plt.imshow(imgg.reshape(28, 28), cmap="gray")
```

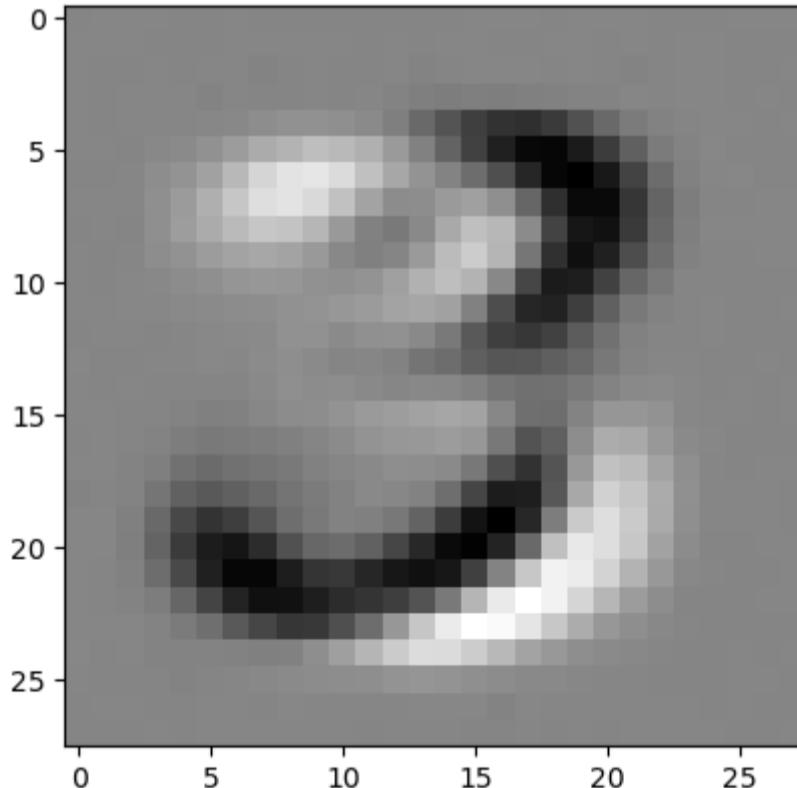
W模长 tensor(27.1404)
-sigma2 tensor(-66.3253)
loss tensor(-66.3253, grad_fn=<NegBackward0>)

epoch: 1, loss: -66.32525634765625
W模长 tensor(35.3270)
-sigma2 tensor(-2667.1165)
loss tensor(-2667.1165, grad_fn=<NegBackward0>)

epoch: 201, loss: -2667.116455078125
W模长 tensor(207.4726)
-sigma2 tensor(-234531.9062)
loss tensor(-234531.9062, grad_fn=<NegBackward0>)

epoch: 401, loss: -234531.90625
W模长 tensor(1959.9087)
-sigma2 tensor(-21430828.)
loss tensor(-21430828., grad_fn=<NegBackward0>)

epoch: 600, loss: -21430828.0
done!



方案二：SGD+输出归一化+适当调大学习率

In [43]: D=torch.tensor(784.)

```
class mymodel(nn.Module):
    def __init__(self):
        super(mymodel, self).__init__()
        self.W=nn.Parameter(torch.randn((784, 1), dtype=torch.float), requires_grad=True)
    def forward(self, x):
        return x.mm(self.W)/D

model=mymodel()
loss_fn=nn.MSELoss()
learning_rate=0.01
optimizer=torch.optim.SGD(model.parameters(), lr=learning_rate)
```

```

num_epoches=600

for epoch in range(num_epoches):
    outputs=model(inputs)
    r=(torch.sum(torch.square(model.W))-800)**2
    loss=-loss_fn(outputs, torch.zeros(811,1))

    # 更新参数
    optimizer.zero_grad()
    loss.backward()
    # nn.utils.clip_grad_norm_(parameters=model.parameters(), max_norm=100, norm_type=2)
    optimizer.step()

    # 每隔 400 次训练输出一次总损失
    if epoch%400==0 or epoch==num_epoches-1:
        with torch.no_grad():
            print("W模长",torch.sqrt(torch.sum(torch.square(model.W))))
            print("-sigma2",-loss_fn(outputs, torch.zeros(811,1)))
            print("loss",loss)
            print("\n")
            print('epoch: {}, loss: {}'.format(epoch+1, loss))

print('done!')
for p in model.parameters():
    np.set_printoptions(suppress=True)
    np.set_printoptions(precision=3)
    imgg=np.copy(p.detach().numpy())

    # print(np.square(imgg).T)
    plt.imshow(imgg.reshape(28,28), cmap="gray")

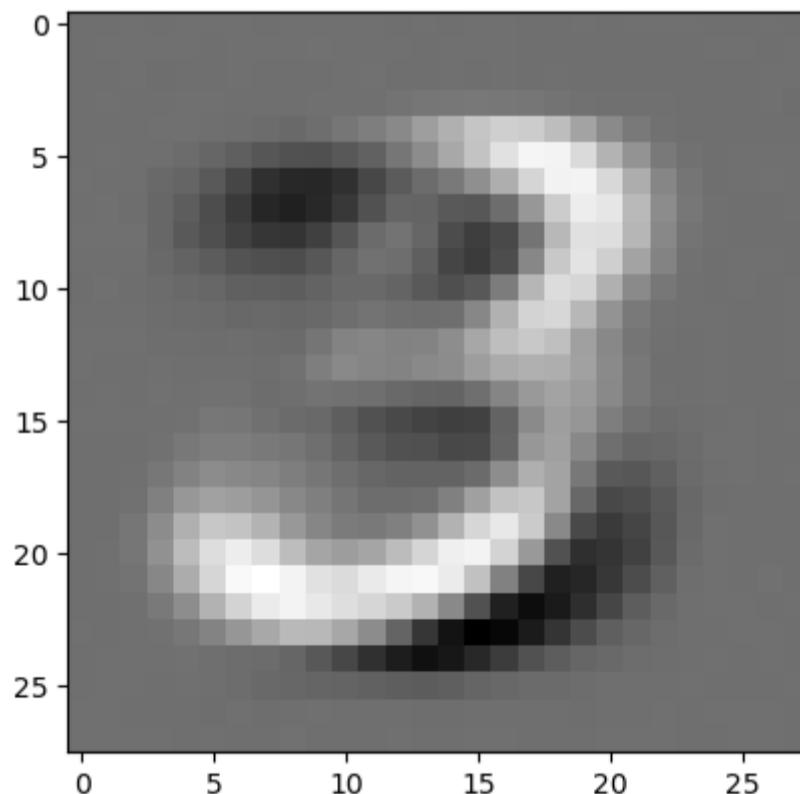
```

W模长 tensor(28.2106)
 -sigma2 tensor(-7.3217)
 loss tensor(-7.3217, grad_fn=<NegBackward0>)

epoch: 1, loss: -7.321678161621094
 W模长 tensor(225.7512)
 -sigma2 tensor(-28362.2285)
 loss tensor(-28362.2285, grad_fn=<NegBackward0>)

epoch: 401, loss: -28362.228515625
 W模长 tensor(2290.3401)
 -sigma2 tensor(-3043501.2500)
 loss tensor(-3043501.2500, grad_fn=<NegBackward0>)

epoch: 600, loss: -3043501.25
 done!



方案三：SGD+梯度裁剪

```

In [44]: class mymodel(nn.Module):
    def __init__(self):
        super(mymodel, self).__init__()
        self.W=nn.Parameter(torch.randn((784,1), dtype=torch.float), requires_grad=True)
    def forward(self, x):
        return x.mm(self.W)

model=mymodel()
loss_fn=nn.MSELoss()
learning_rate=0.01
optimizer=torch.optim.SGD(model.parameters(), lr=learning_rate)

```

```

num_epochs=600

for epoch in range(num_epochs):
    outputs=model(inputs)
    r=(torch.sum(torch.square(model.W))-800)**2
    loss=-loss_fn(outputs, torch.zeros(811,1))

    # 更新参数
    optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(parameters=model.parameters(), max_norm=100, norm_type=2)
    optimizer.step()

    # 每隔 400 次训练输出一次总损失
    if epoch%400==0 or epoch==num_epochs-1:
        with torch.no_grad():
            print("W模长", torch.sqrt(torch.sum(torch.square(model.W))))
            print("-sigma2", -loss_fn(outputs, torch.zeros(811,1)))
            print("loss", loss)
            print("\n")
            print('epoch: {}, loss: {}'.format(epoch+1, loss))

print('done!')
for p in model.parameters():
    np.set_printoptions(suppress=True)
    np.set_printoptions(precision=3)
    imgg=np.copy(p.detach().numpy())

    # print(np.square(imgg).T)
    plt.imshow(imgg.reshape(28, 28), cmap="gray")

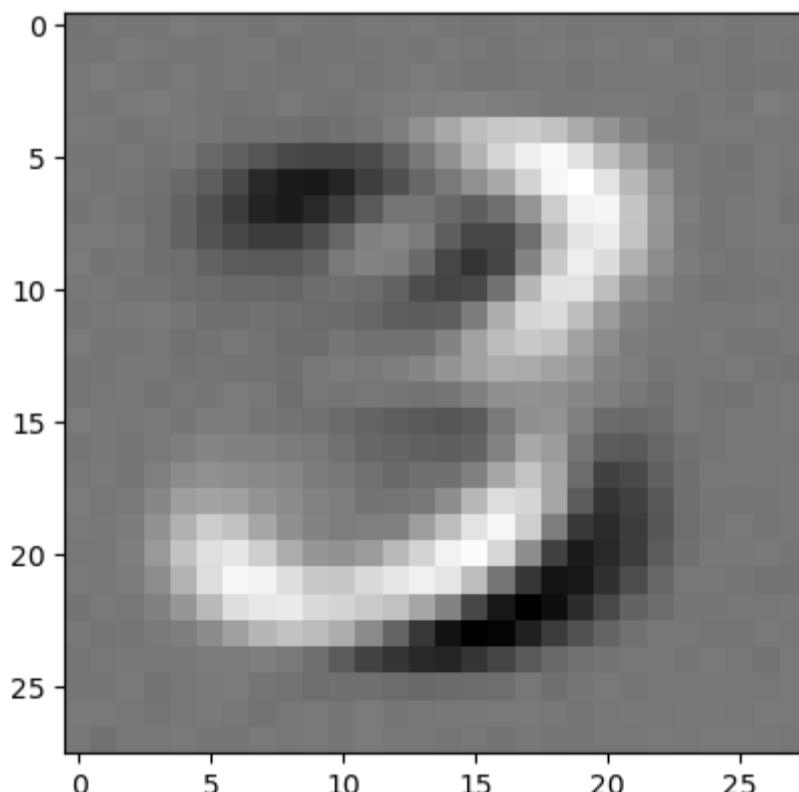
```

W模长 tensor(28.7351)
 -sigma2 tensor(-3512027.7500)
 loss tensor(-3512027.7500, grad_fn=<NegBackward0>)

epoch: 1, loss: -3512027.75
 W模长 tensor(403.9850)
 -sigma2 tensor(-5.9938e+10)
 loss tensor(-5.9938e+10, grad_fn=<NegBackward0>)

epoch: 401, loss: -59937562624.0
 W模长 tensor(602.5970)
 -sigma2 tensor(-1.3401e+11)
 loss tensor(-1.3401e+11, grad_fn=<NegBackward0>)

epoch: 600, loss: -134007734272.0
 done!



方案四：超级小的学习率+较大的训练次数

```

In [45]: class mymodel(nn.Module):
    def __init__(self):
        super(mymodel, self).__init__()
        self.W=nn.Parameter(torch.randn((784, 1), dtype=torch.float), requires_grad=True)
    def forward(self, x):
        return x.mm(self.W)

model=mymodel()
loss_fn=nn.MSELoss()
learning_rate=0.00000001
optimizer=torch.optim.SGD(model.parameters(), lr=learning_rate)

```

```

num_epochs=6000

for epoch in range(num_epochs):
    outputs=model(inputs)
    r=(torch.sum(torch.square(model.W))-800)**2
    loss=-loss_fn(outputs, torch.zeros(811,1))

    # 更新参数
    optimizer.zero_grad()
    loss.backward()
    # nn.utils.clip_grad_norm_(parameters=model.parameters(), max_norm=100, norm_type=2)
    optimizer.step()

    # 每隔 1000 次训练输出一次总损失
    if epoch%1000==0 or epoch==num_epochs-1:
        with torch.no_grad():
            print("W模长", torch.sqrt(torch.sum(torch.square(model.W))))
            print("-sigma2", -loss_fn(outputs, torch.zeros(811,1)))
            print("loss", loss)
            print("\n")
            print('epoch: {}, loss: {}'.format(epoch+1, loss))

print('done!')
for p in model.parameters():
    np.set_printoptions(suppress=True)
    np.set_printoptions(precision=3)
    imgg=np.copy(p.detach().numpy())

    # print(np.square(imgg).T)
    plt.imshow(imgg.reshape(28,28), cmap="gray")

```

W模长 tensor(28.7779)
 -sigma2 tensor(-3164420.)
 loss tensor(-3164420., grad_fn=<NegBackward0>)

epoch: 1, loss: -3164420.0
 W模长 tensor(991.8088)
 -sigma2 tensor(-3.5468e+11)
 loss tensor(-3.5468e+11, grad_fn=<NegBackward0>)

epoch: 1001, loss: -354678538240.0
 W模长 tensor(1593676.7500)
 -sigma2 tensor(-9.2966e+17)
 loss tensor(-9.2966e+17, grad_fn=<NegBackward0>)

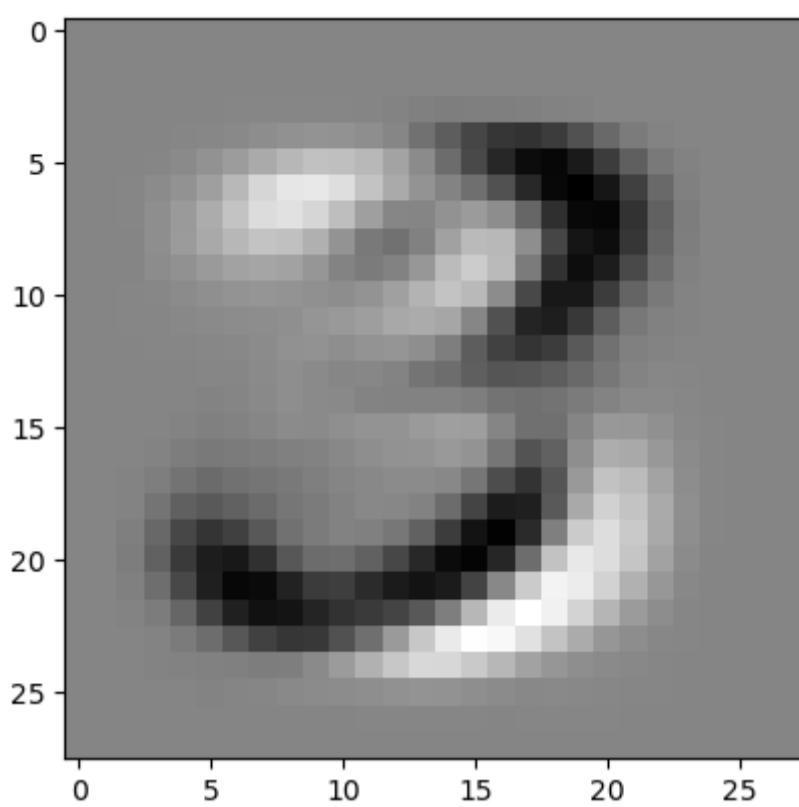
epoch: 2001, loss: -9.296612018209423e+17
 W模长 tensor(2.6160e+09)
 -sigma2 tensor(-2.5054e+24)
 loss tensor(-2.5054e+24, grad_fn=<NegBackward0>)

epoch: 3001, loss: -2.505365587188322e+24
 W模长 tensor(4.2956e+12)
 -sigma2 tensor(-6.7553e+30)
 loss tensor(-6.7553e+30, grad_fn=<NegBackward0>)

epoch: 4001, loss: -6.75531487948381e+30
 W模长 tensor(7.0537e+15)
 -sigma2 tensor(-inf)
 loss tensor(-inf, grad_fn=<NegBackward0>)

epoch: 5001, loss: -inf
 W模长 tensor(1.1497e+19)
 -sigma2 tensor(-inf)
 loss tensor(-inf, grad_fn=<NegBackward0>)

epoch: 6000, loss: -inf
 done!



方案五：Adam+归一化+正则化

```
In [46]: inputs3=(inputs-inputs.mean(axis=0))/255
class mymodel(nn.Module):
    def __init__(self):
        super(mymodel, self).__init__()
        self.W=nn.Parameter(torch.randn((784, 1), dtype=torch.float), requires_grad=True)
    def forward(self, x):
        return x.mm(self.W)

model=mymodel()
loss_fn=nn.MSELoss()
learning_rate=0.001
optimizer=torch.optim.SGD(model.parameters(), lr=learning_rate)

num_epochs=1200

for epoch in range(num_epochs):
    outputs=model(inputs3)
    r=(torch.sum(torch.square(model.W))-800)**2
    loss=-loss_fn(outputs, torch.zeros(811, 1))+0.4*r

    # 更新参数
    optimizer.zero_grad()
    loss.backward()
    # nn.utils.clip_grad_norm_(parameters=model.parameters(), max_norm=100, norm_type=2)
    optimizer.step()

    # 每隔 400 次训练输出一次总损失
    if epoch%400==0 or epoch==num_epochs-1:
        with torch.no_grad():
            print("W模长", torch.sqrt(torch.sum(torch.square(model.W))))
            print("-sigma2", -loss_fn(outputs, torch.zeros(811, 1)))
            print("loss", loss)
            print("\n")
            print('epoch: {}, loss: {}'.format(epoch+1, loss))

print('done!')
for p in model.parameters():
    np.set_printoptions(suppress=True)
    np.set_printoptions(precision=3)
    imgg=np.copy(p.detach().numpy())

    # print(np.square(imgg).T)
    plt.imshow(imgg.reshape(28, 28), cmap="gray")
```

```
W模长 tensor(28.6550)
-sigma2 tensor(-29.6532)
loss tensor(44.2874, grad_fn=<AddBackward0>)
```

```
epoch: 1, loss: 44.287376403808594
W模长 tensor(33.2829)
-sigma2 tensor(-2649.1194)
loss tensor(30232.0488, grad_fn=<AddBackward0>)
```

```
epoch: 401, loss: 30232.048828125
W模长 tensor(33.3734)
-sigma2 tensor(-2771.9966)
loss tensor(36821.8398, grad_fn=<AddBackward0>)
```

```
epoch: 801, loss: 36821.83984375
W模长 tensor(22.0309)
-sigma2 tensor(-5689.8223)
loss tensor(9645.6338, grad_fn=<AddBackward0>)
```

```
epoch: 1200, loss: 9645.6337890625
done!
```

