Tutorial_3

Andrea B

2023-10-07

Álgebra lineal en matrices de n dimensiones.

En este tutorial, se utilizará una descomposición matricial del álgebra lineal, la descomposición de valores singulares, para generar una aproximación comprimida de una imagen.Se usará la imagen del módulo scipy.misc:

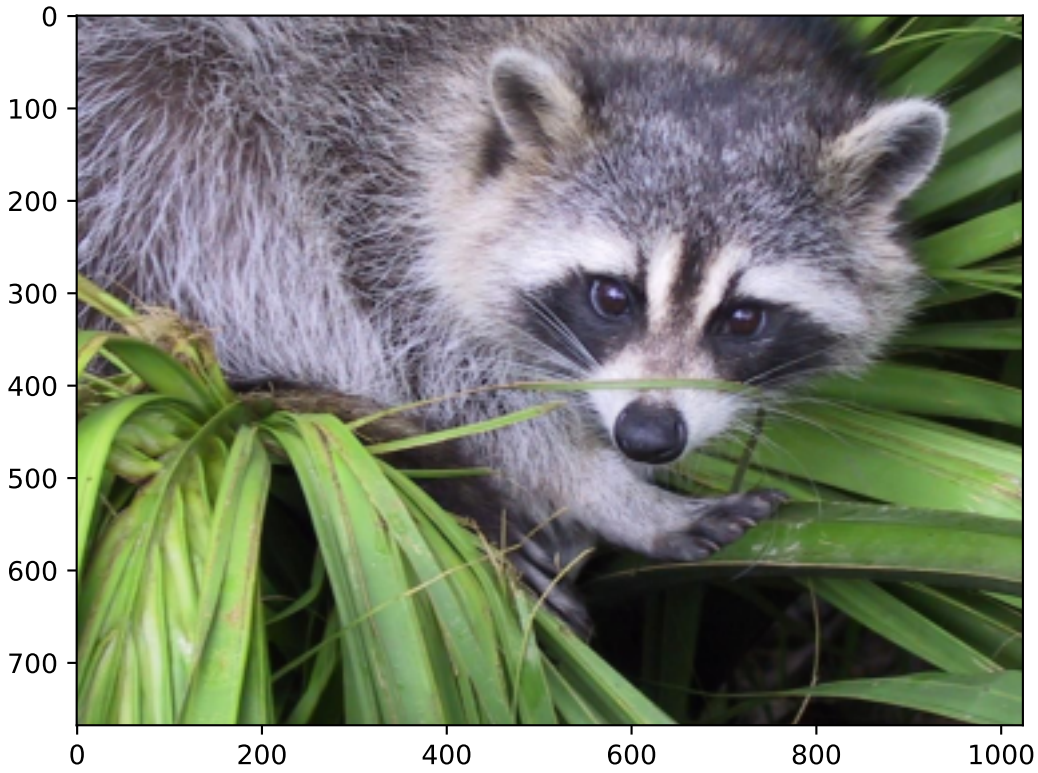
```
import scipy.datasets
img= scipy.datasets.face()
```

```
import numpy as np
type(img)
```

```
## <class 'numpy.ndarray'>
```

Podemos ver la imagen usando la función y el comando especial Python, para mostrar gráficos en línea:

```
import matplotlib.pyplot as plt
plt.imshow(img)
plt.show()
```



Propiedades de forma, eje y matriz

Tenga en cuenta que, en álgebra lineal, la dimensión de un vector se refiere al número de entradas en una matriz. Primero, verificamos la forma de los datos en la matriz. Dado que esta imagen es bidimensional (los píxeles de la imagen forman un rectángulo), podríamos esperar que una matriz bidimensional la represente (una matriz). Sin embargo, usar la shape propiedad de esta matriz NumPy nos da un resultado diferente.

```
img.shape
```

```
## (768, 1024, 3)
```

La salida es una tupla con tres elementos, lo que significa que se trata de una matriz tridimensional. De hecho, como se trata de una imagen en color y hemos utilizado la imread función para leerla, los datos están organizados en tres matrices 2D, que representan canales de color (en este caso, rojo, verde y azul - RGB). Puedes ver esto mirando la forma de arriba: indica que tenemos una matriz de 3 matrices, cada una con una forma de 768x1024.

Además, usando la ndim propiedad de esta matriz, podemos ver que:

```
img.ndim
```

```
## 3
```

NumPy se refiere a cada dimensión como un *eje* . Por cómo imread funciona, el *primer índice en el tercer eje* son los datos del píxel rojo de nuestra imagen. Podemos acceder a esto usando la sintaxis:

```
img[:, :, 0]
```

```
## array([[121, 138, 153, ..., 119, 131, 139],
##        [ 89, 110, 130, ..., 118, 134, 146],
##        [ 73,  94, 115, ..., 117, 133, 144],
##        ...,
##        [ 87,  94, 107, ..., 120, 119, 119],
##        [ 85,  95, 112, ..., 121, 120, 120],
##        [ 85,  97, 111, ..., 120, 119, 118]], dtype=uint8)
```

En el resultado anterior, podemos ver que cada valor es un valor entero entre 0 y 255, que representa el nivel de rojo en cada píxel de la imagen correspondiente.Como era de esperar, esta es una matriz de 768x1024:

```
img[:, :, 0].shape
```

```
## (768, 1024)
```

Dado que vamos a realizar operaciones de álgebra lineal con estos datos, podría ser más interesante tener números reales entre 0 y 1 en cada entrada de las matrices para representar los valores RGB. Podemos hacerlo configurando.

```
img_array = img / 255
```

Esta operación, que divide una matriz por un escalar, funciona gracias a las reglas de transmisión de NumPy .Puede comprobar que lo anterior funciona haciendo algunas pruebas; por ejemplo, consultando sobre los valores máximos y mínimos para esta matriz:

```
img_array.max(), img_array.min()
```

```
## (1.0, 0.0)
```

o comprobando el tipo de datos en la matriz:

```
img_array.dtype
```

```
## dtype('float64')
```

Tenga en cuenta que podemos asignar cada canal de color a una matriz separada usando la sintaxis de corte:

```
red_array = img_array[:, :, 0]
green_array = img_array[:, :, 1]
blue_array = img_array[:, :, 2]
```

Operaciones sobre un eje.

Es posible utilizar métodos del álgebra lineal para aproximar un conjunto de datos existente. Aquí, usaremos la svd (descomposición de valores singulares) para intentar reconstruir una imagen que utiliza menos información de valores singulares que la original, conservando al mismo tiempo algunas de sus características.

Para continuar, importe el submódulo de álgebra lineal desde NumPy:

```
from numpy import linalg
```

Para extraer información de una matriz determinada, podemos usar el SVD para obtener 3 matrices que se pueden multiplicar para obtener la matriz original.De la teoría del álgebra lineal, dada una matriz A se puede calcular el siguiente producto:

Observe que podemos usar el operador (el operador de multiplicación de matrices para matrices NumPy para esto:

```
img_gray = img_array @ [0.2126, 0.7152, 0.0722]
```

Ahora img_gray tiene forma.

```
img_gray.shape
```

```
## (768, 1024)
```

Para ver si esto tiene sentido en nuestra imagen, debemos usar un mapa de colores matplotlib correspondiente al color que deseamos ver en nuestra imagen (de lo contrario,matplotlib usaremos por defecto un mapa de colores que no corresponde a los datos reales).

En nuestro caso, nos estamos aproximando a la parte en escala de grises de la imagen, por lo que usaremos el mapa de colores gray:

```
plt.imshow(img_gray, cmap="gray")
plt.show()
```

Ahora, aplicando la función linalg.svd a esta matriz, obtenemos la siguiente descomposición:

```
U, s, Vt = linalg.svd(img_gray)
```

Comprobemos que esto es lo que esperábamos:

```
U.shape, s.shape, Vt.shape
```

```
## ((768, 768), (768,), (1024, 1024))
```

Da como resultado un valueError. Esto sucede porque tener una matriz unidimensional para S, en este caso, es mucho más económico en la práctica que construir una matriz diagonal con los mismos datos. Para reconstruir la matriz original, podemos reconstruir la matriz diagonal. con los elementos de S en su diagonal y con las dimensiones adecuadas para multiplicar: en nuestro caso, debe ser 768x1024 ya que u es 768x768 y vt es 1024x1024. Para sumar los valores singulares a la diagonal de sigma, usaremos la función:

```
import numpy as np
Sigma = np.zeros((U.shape[1], Vt.shape[0]))
np.fill_diagonal(Sigma, s)
```

Aproximación

El módulo linalg incluye una norm función que calcula la norma de un vector o matriz representado en una matriz NumPy. Por ejemplo, a partir de la explicación anterior de SVD, esperaríamos que la norma de la diferencia entre img_gray el producto SVD reconstruido y el producto SVD fuera pequeña. Como era de esperar, deberías ver algo como:

```
linalg.norm(img_gray - U @ Sigma @ Vt)
```

```
## 1.5985090441460665e-12
```

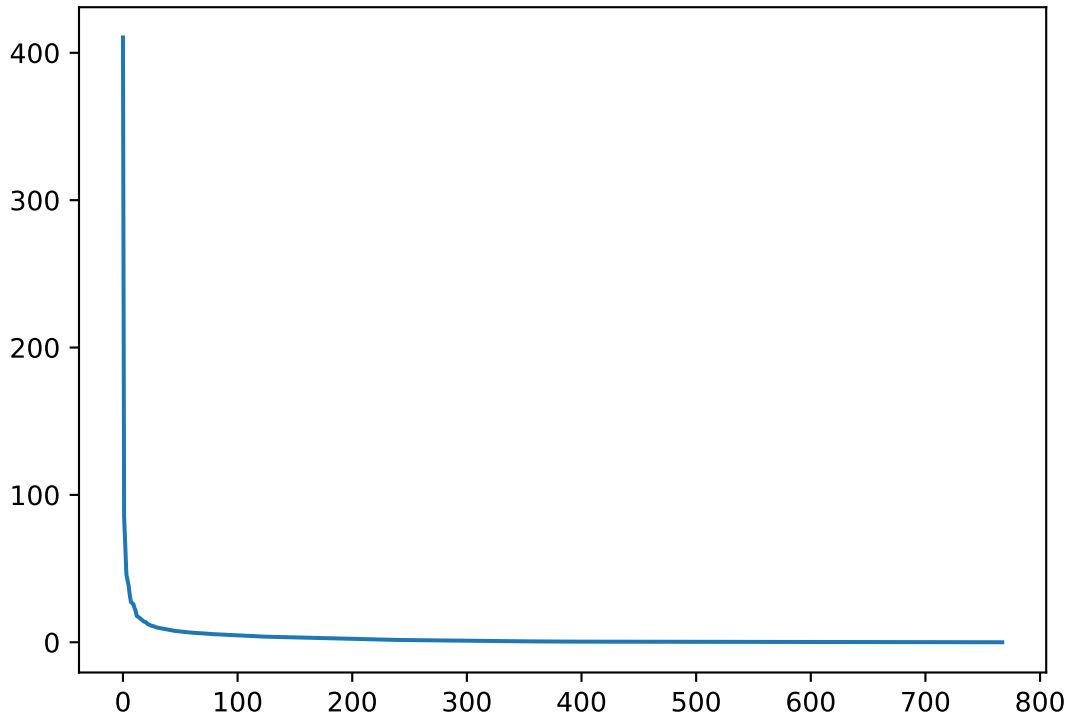
También podríamos haber usado la función numpy.allclose para asegurarnos de que el producto reconstruido esté, de hecho, cerca de nuestra matriz original (la diferencia entre las dos matrices es pequeña):

```
np.allclose(img_gray, U @ Sigma @ Vt)
```

```
## True
```

Para ver si una aproximación es razonable, podemos comprobar los valores en S:

```
plt.plot(s)
plt.show()
```



En el gráfico, podemos ver que aunque tenemos 768 valores singulares en S, la mayoría de ellos (después de la entrada número 150 aproximadamente) son bastante pequeños. Por lo tanto, podría tener sentido utilizar sólo

la información relacionada con los primeros (digamos, 50) *valores singulares* para construir una aproximación más económica a nuestra imagen.

La idea es considerar todos los k valores singulares excepto los primeros en sigma (que son los mismos que en s) como ceros, manteniendo u e vt intacto, y calcular el producto de estas matrices como aproximación.

Por ejemplo, si elegimos:

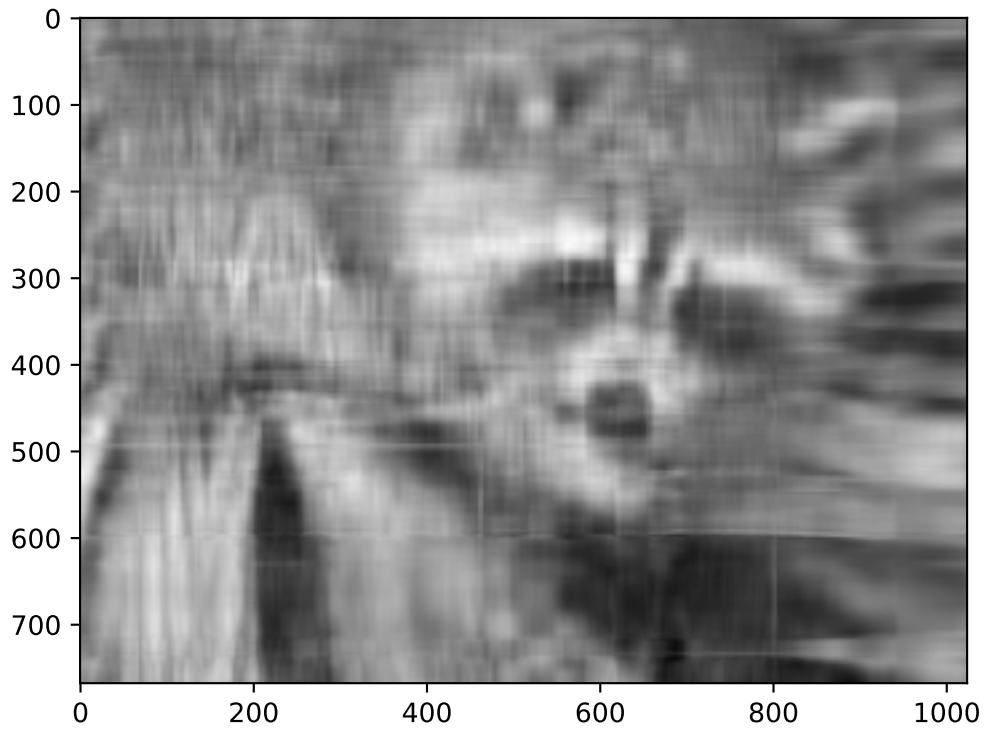
```
k = 10
```

Podemos construir la aproximación haciendo:

```
approx = U @ Sigma[:, :k] @ Vt[:k, :]
```

Tenga en cuenta que tuvimos que usar solo las primeras k filas de vt, ya que todas las demás filas se multiplicarían por los ceros correspondientes a los valores singulares que eliminamos de esta aproximación.

```
plt.imshow(approx, cmap="gray")
plt.show()
```



Aplicando a todos los colores.

Ahora queremos hacer el mismo tipo de operación, pero con los tres colores. Nuestro primer instinto podría ser repetir la misma operación que hicimos anteriormente para cada matriz de color individualmente. *Sin embargo, la transmisión* de NumPy se encarga de esto por nosotros.

Si nuestra matriz tiene más de dos dimensiones, entonces el SVD se puede aplicar a todos los ejes a la vez. Sin embargo, las funciones de álgebra lineal en NumPy esperan ver una matriz de la forma , donde el primer eje representa el número de matrices en la pila. En nuestro caso:

```
img_array.shape
```

```
## (768, 1024, 3)
```

Indica que el eje se reordenará de manera que la forma final de la matriz transpuesta se reordenará de acuerdo con los índices .Veamos cómo funciona esto para nuestra matriz:

```
img_array_transposed = np.transpose(img_array, (2, 0, 1))
img_array_transposed.shape
```

```
## (3, 768, 1024)
```

Ahora estamos listos para aplicar la SVD:

```
U, s, Vt = linalg.svd(img_array_transposed)
```

Finalmente, para obtener la imagen aproximada completa, necesitamos volver a ensamblar estas matrices en la aproximación. Ahora, tenga en cuenta que:

```
U.shape, s.shape, Vt.shape
```

```
## ((3, 768, 768), (3, 768), (3, 1024, 1024))
```

Productos con matrices n-dimensionales.

Si ha trabajado antes con matrices de una o dos dimensiones en NumPy, puede usar numpy (o el `@`operador) indistintamente. Sin embargo, para matrices de n dimensiones, funcionan de maneras muy diferentes.

Ahora, para construir nuestra aproximación, primero debemos asegurarnos de que nuestros valores singulares estén listos para la multiplicación, por lo que construimos nuestra matriz de manera similar a lo que hicimos antes. La matriz debe tener dimensiones . Para sumar los valores singulares a la diagonal de , usaremos nuevamente la función , usando cada una de las 3 filas como diagonal para cada una de las 3 matrices en :

```
Sigma = np.zeros((3, 768, 1024))
for j in range(3):
    np.fill_diagonal(Sigma[j, :, :], s[j, :])
```

Ahora, si deseamos reconstruir el SVD completo (sin aproximación), podemos hacer:

```
reconstructed = U @ Sigma @ Vt
```

Tenga en cuenta que:

```
reconstructed.shape
```

```
## (3, 768, 1024)
```

La imagen reconstruida debe ser indistinguible de la original, excepto por diferencias debidas a errores de punto flotante de la reconstrucción. Recuerde que nuestra imagen original constaba de valores de punto flotante en el rango . La acumulación de error de punto flotante de la reconstrucción puede dar como resultado valores ligeramente fuera de este rango original:

```
reconstructed.min(), reconstructed.max()
```

```
## (-6.013798400233972e-15, 1.0000000000000062)
```

Dado que se esperan valores en el rango, podemos utilizar clip para eliminar el error de punto flotante:

```
reconstructed = np.clip(reconstructed, 0, 1)
plt.imshow(np.transpose(reconstructed, (1, 2, 0)))
plt.show()
```

De hecho, realiza este recorte bajo el capó, por lo que si omite la primera línea en la celda del código anterior, ahora, para hacer la aproximación, debemos elegir solo los primeros k valores singulares para cada canal de color. Esto se puede hacer usando la siguiente sintaxis:

```
approx_img = U @ Sigma[..., :k] @ Vt[..., :k, :]
```

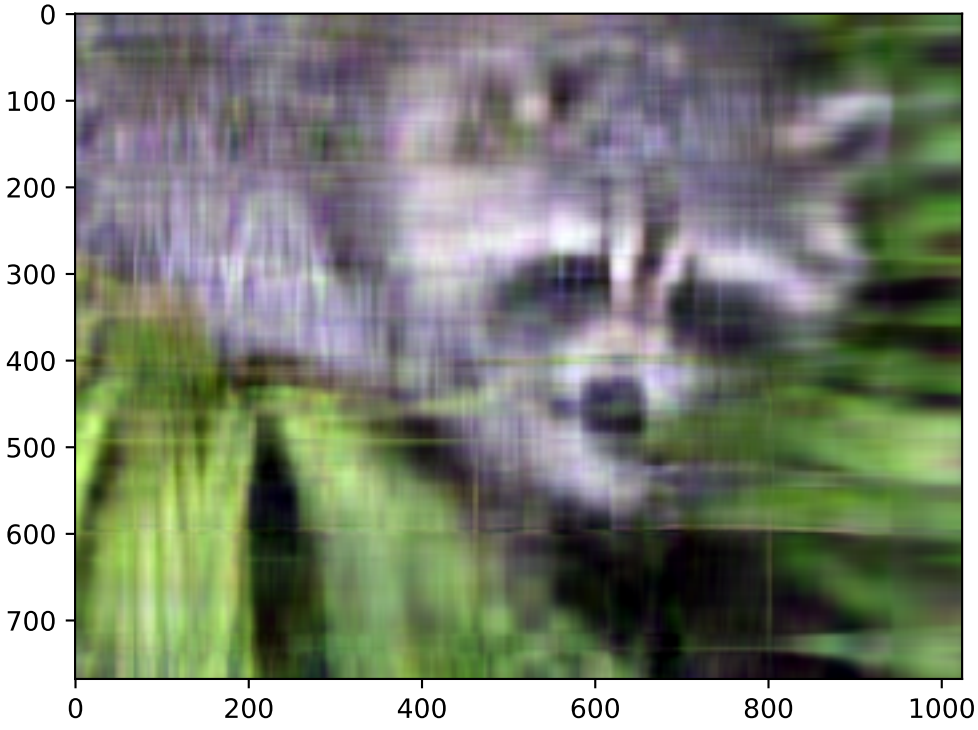
Puedes ver que hemos seleccionado solo los primeros k componentes del último eje (esto significa que hemos usado solo las primeras k columnas de cada una de las tres matrices en la pila), y que hemos seleccionado solo los primeros k componentes en el segundo. -al último eje de vt (esto significa que hemos seleccionado solo las primeras k filas de cada matriz en la pila vt y todas las columnas). Si no está familiarizado con la sintaxis de puntos suspensivos, es un marcador de posición para otros ejes. Ahora:

```
approx_img.shape
```

```
## (3, 768, 1024)
```

Que no es la forma adecuada para mostrar la imagen. Finalmente, reordenando los ejes a nuestra forma original de , podemos ver nuestra aproximación:

```
plt.imshow(np.transpose(approx_img, (1, 2, 0)))
plt.show()
```



Aunque la imagen no es tan nítida, utilizando una pequeña cantidad de k valores singulares (en comparación con el conjunto original de 768 valores), podemos recuperar muchas de las características distintivas de esta imagen.

Palabras finales

Por supuesto, este no es el mejor método para *aproximar* una imagen. Sin embargo, de hecho, hay un resultado en álgebra lineal que dice que la aproximación que construimos anteriormente es la mejor que podemos obtener de la matriz original en términos de la norma de la diferencia.