ENTREGA 4

ÁLGEBRA LINEAL COMPUTACIONAL

Grado en Matemáticas

Autor

Emilio Domínguez, Sergio Mínguez y Álvaro Inclán

Madrid, 31/10/2024

Entregable: Compresión de Imágenes utilizando Randomized SVD

1 Introducción

En este caso práctico, exploraremos la compresión de imágenes mediante la técnica de descomposición en valores singulares aleatorizada (*Randomized SVD*). La compresión de imágenes es una aplicación importante para la reducción de la cantidad de datos a almacenar, preservando la mayor parte de la información visual. El *Randomized SVD* nos permite realizar esta compresión de manera eficiente para imágenes de gran tamaño.

La tarea principal consiste en descomponer una imagen en componentes principales, seleccionar las más relevantes, y reconstruir la imagen comprimida a partir de estos componentes, minimizando la pérdida de calidad visual.

2 Objetivo

El objetivo de este caso práctico es aplicar la técnica de *Randomized SVD* para comprimir una imagen representada como una matriz A. La descomposición SVD permite descomponer la matriz en tres matrices:

$$A \approx U\Sigma V^T$$

Donde:

- U es una matriz que contiene los vectores singulares de la imagen.
- Σ es una matriz diagonal que contiene los valores singulares.
- V^T contiene los vectores singulares de la imagen transpuesta.

Al reducir la cantidad de valores singulares utilizados en la reconstrucción de la imagen, podemos disminuir el tamaño de los datos sin perder demasiada información visual.

3 Planteamiento del Problema

El objetivo del caso práctico es implementar un algoritmo en Python que realice los siguientes pasos:

1. Cargar una imagen y convertirla en una matriz de valores de intensidad de píxeles.
2. Aplicar la descomposición SVD aleatorizada (*Randomized SVD*) a la matriz de la imagen.
3. Seleccionar los k valores singulares más significativos para reconstruir la imagen comprimida.
4. Evaluar la calidad de la compresión comparando la imagen comprimida con la original.

4 Datos

Los alumnos deberán seleccionar una imagen de su elección para trabajar con el caso práctico. Pueden utilizar cualquier formato de imagen (JPG, PNG, etc.) y convertirla a escala de grises para simplificar el procesamiento.

5 Tareas a realizar

Para resolver este caso práctico, los alumnos deben implementar las siguientes tareas:

5.1 Carga y preprocesamiento de la imagen

Cargar la imagen seleccionada y convertirla a escala de grises, si es necesario. A continuación, convertir la imagen en una matriz de valores de intensidad de píxeles.

5.2 Aplicación de Randomized SVD

Utilizar la función `randomized_svd` de la biblioteca `scikit-learn` para aplicar *Randomized SVD* a la matriz de la imagen. La función debe retornar las tres matrices U, Σ y V^T.

5.3 Compresión y reconstrucción de la imagen

Reconstruir la imagen comprimida utilizando únicamente los k valores singulares más grandes. Los alumnos deberán seleccionar k y justificar su elección en función de la calidad visual de la imagen reconstruida y la cantidad de compresión lograda.

5.4 Evaluación de la compresión

Evaluar el error de reconstrucción utilizando la norma de Frobenius para medir la diferencia entre la imagen original y la comprimida:

$$\text{Error} = \frac{\|A - A_{approx}\|_F}{\|A\|_F}$$

6 Entregables

Los alumnos deberán entregar un informe que incluya:

- El código Python desarrollado para cargar la imagen, aplicar *Randomized SVD*, y reconstruir la imagen comprimida.
- Gráficas de la imagen original y la imagen comprimida.
- Análisis del error de reconstrucción y su interpretación.
- Reflexiones sobre el número de valores singulares k seleccionados y su impacto en la calidad de la imagen y el tamaño comprimido.

7 Recursos

Para completar este caso práctico, los alumnos pueden hacer uso de las siguientes bibliotecas de Python:

- `NumPy` para el manejo de matrices y operaciones numéricas.
- `Matplotlib` para la visualización de la imagen original y la comprimida.
- `scikit-learn` para la implementación de *Randomized SVD*.
- `Pillow` para cargar y preprocesar imágenes.

8 Conclusión

Este caso práctico permitirá a los alumnos entender cómo aplicar la técnica de *Randomized SVD* en el contexto de la compresión de imágenes. La compresión de imágenes es una aplicación crucial en áreas como el almacenamiento de grandes bases de datos visuales y la transmisión eficiente de datos a través de la web. Al reducir la dimensionalidad de los datos, podemos mejorar la eficiencia sin comprometer significativamente la calidad.

Índice

1. Resolución

1.1. Introducción

Nuestro trabajo se centra en la compresión de la imagen con la tecnica de descomposición en valores singulares aleatorizada.

Es una manera de reducir el espacio que ocupa una imagen pero sin perder gran parte de la información visual. Lo que buscamos es comprimir imagenes de manera muy eficiente. El ordenador esta perdiendo informacion pero nosotros no vamos a perder tanta información a nivel visual, va a ser complicado diferenciar entre las dos imagenes pero el ordenador si que nota la diferencia de tamaño (a nivel de memoria)

Para la realización del trabajo vamos a utilizar la liberia sklearn. En esta libreria tenemos la función randomized_svd que nos descompondra la matriz A en lo que buscamos. Ahora intentaremos explicar que hace esta función.

Formulación del Problema

Este problema es hacer un pequeño retoque a el metodo clasico SVD. Para ello lo primero que hacemos es plantear una matriz $A \in \mathbb{R}^{mxn}$ y vamos a realizar una reducción de la dimensionalidad. Despues tomamos un valor K con K $\ll min(m,n)$ para pasar de una matriz $A \in \mathbb{R}^{mxn}$ a una matriz $Y \in \mathbb{R}^{mxk}$. Aqui esta lo mas importante del metodo para pasar de una matriz de dimensiones considerables a una matriz de dimensiones mas manejables.
Esto lo hacemos a través de una matriz de proyecciones $G \in \mathbb{R}^{nxk}$. Entonces creamos la matriz Y de la siguiente manera: $Y = AG \in \mathbb{R}^{mxk}$ Esta es la parte más importante del método debido a que baja el computo operacional de $O(mn^2)$ con el método SVD clásico a un coste operacional del orden de $O(mnlog(k))$.

Hay muchas maneras de hacer una reducción de la dimensionalidad, en la entrega previa vimos y programamos el metodo Count Sketching. En este ejercicio no sabemos que método realiza el programa para la reducción de la dimensionalidad. Hemos analizado el código contigo en clase y no entendemos que usa para la reducción, creemos que es una proyección simple sobre la dimension que tenemos.
Aunque no sepamos que método realiza la función para la reducción de la dimensionalidad hemos confiado en el programa. Las librerías de python se crean de una manera general y tenemos el código para leerlo. Confiamos que es una librería que se ha comprobado y la comunidad ha aceptado como

una solución buena para poder utilizarla.

Cuando ya tenemos la matriz $Y \in \mathbb{R}^{mxk}$ podemos aplicar el método clásico de SVD. Para ello necesito descomponer la matriz Y en tres matrices distintas

$$Y = U_Y \Sigma_Y V_Y^T$$

- $U_Y \in \mathbb{R}^{mxm}$ es una matriz que contiene los vectores singulares de la imagen.
- $\Sigma_Y \in \mathbb{R}^{mxk}$ es una matriz diagonal que contiene los valores singulares.
- $V_Y^T \in \mathbb{R}^{kxk}$ contiene los vectores singulares de la imagen transpuesta.

Cuando ya tenemos la matriz separada en estas matrices lo importante en lo que nos fijamos es en la matriz Σ_Y. Esta matriz tiene los valores singulares y esta ordenada en función del tamaño de los valores singulares. En el problema vamos a olvidarnos de los valores singulares con valores menores. Aquí es cuando estamos reduciendo lo que ocupa en el ordenador, estamos cambiando esos valores a 0 y quedandonos con los valores singulares más altos ya que contienen la gran parte de la información de la matriz Y.

La parte final del proceso es volver a una matriz del tamaño original, para ello la función genera una matriz G^* para poder encontrar una descomposición aproximada de la descomposición SVD. No sabemos como se encuentra esta matriz para la reconstrucción de la matriz A pero confiamos en la comunidad y el método que realiza.

Esa es la base de nuestro problema y lo que tenemos que considerar es cuántos valores singulares cogemos para mantener una claridad visual lo suficientemente buena para que se sigua entendiendo la imagen pero que la imagen ocupe menos espacio de memoria.

En esta época digital es muy importante poder comprimir imagenes para el desarrorllo de paginas web, estamos consiguiendo una optimizacion en el rendimiento y en el almacenamiento de nuestros archivos.

El problema viene al final con el equilibrio entre la reducción de la dimensionalidad, cuántos valores singulares tomamos y la compresión visual de la imagen. En esencia, estamos equilibrando cuanto ocupa de espacio en el ordenador y lo bien que mantiene la imagen original (la calidad que pierde cuando hace el proceso)

El único problema que tenemos es como un ordenador guarda una imagen (en color) con el sistema RGB.

El ordenador guarda la cantidad de rojo, verde y azul en tres matrices distintas pero lo mete en una misma matriz de tamaño $A \in \mathbb{R}^{mxnx3}$, con el último valor haciendo referencia a si es mxnx1 a

el color rojo, mxnx2 a el color azul y mxnx3 a el color verde. Entonces, tenemos tres matrices a la que vamos a aplicarle el método RandomizedSVD por separado.

1.2. Código

A continuación, mostramos el código explicado:

1.2.1. Importación de Bibliotecas

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
from sklearn.utils.extmath import randomized_svd
```

- **PIL (Python Imaging Library)**: Se utiliza para abrir y manejar imágenes.
- **NumPy**: Biblioteca para cálculos numéricos y manejo de matrices.
- **Matplotlib**: Usado para visualizar imágenes.
- **randomized_svd**: Una implementación eficiente de la descomposición en valores singulares (SVD) con un enfoque aleatorio, parte de `scikit-learn`.

1.2.2. Cargar una Imagen en Color

```
def cargar_imagen_color(ruta_imagen):
    imagen = Image.open(ruta_imagen)
    matriz_imagen = np.array(imagen)  # Matriz 3D con RGB
    return matriz_imagen
```

Esta función carga una imagen desde una ruta especificada y la convierte en una matriz 3D (`np.array`) con las dimensiones $(alto, ancho, 3)$, donde los tres canales corresponden a los colores **Rojo, Verde y Azul** (RGB).

1.2.3. Visualizar la Imagen Original

```
plt.imshow(matriz_imagen)
plt.title('Imagen Original en Color')
plt.axis('off')
plt.show()
```

Se utiliza `Matplotlib` para mostrar la imagen cargada en color, y `plt.axis('off')` elimina los ejes para una visualización más limpia.

1.2.4. Aplicar Randomized SVD a un Canal de Color

```
def aplicar_rsvd(matriz_canal, k):
    U, Sigma, VT = randomized_svd(matriz_canal,
    n_components=k, random_state=42)
    return U, Sigma, VT
```

Esta función realiza una descomposición en valores singulares randomized (SVD) sobre un solo canal (Rojo, Verde o Azul) de la imagen. La función `randomized_svd` descompone la matriz `matriz_canal` en tres matrices: U, Σ (valores singulares) y VT, utilizando sólo k componentes principales para permitir la compresión.

1.2.5. Comprimir y Reconstruir la Imagen en Color

```
def comprimir_imagen_color(matriz_imagen, k):
    canales = []
    tamano_comprimido = 0  # Para calcular el tamano
    total de la imagen comprimida

    for i in range(3):  # Iterar sobre los canales R, G, B
        U, Sigma, VT = aplicar_rsvd(matriz_imagen[:, :, i], k)
        matriz_aproximada = np.dot(U, np.dot(np.diag(Sigma), VT))
        canales.append(matriz_aproximada)

        # Calcular tamano de la imagen comprimida para este canal
        tamano_comprimido += U.size + Sigma.size + VT.size

    # Combina los canales comprimidos
    img_comprimida = np.stack(canales, axis=2)
    return img_comprimida, tamano_comprimido
```

Esta función comprime cada uno de los tres canales de color (R, G, B) de la imagen utilizando **Randomized SVD**:

- **Para cada canal**: Se aplica la función `aplicar_rsvd` para descomponer el canal en sus componentes U, Σ y VT.
- **Reconstrucción**: La imagen aproximada de cada canal se reconstruye mediante el producto de matrices $U \cdot \text{diag}(\Sigma) \cdot VT$, reteniendo sólo los primeros k valores singulares.

- **Cálculo del Tamaño Comprimido**: Se calcula el tamaño total de la compresión sumando el número de elementos en las matrices U, Σ y VT para cada canal.

1.2.6. Visualizar la Imagen Comprimida

```
plt.imshow(np.clip(imagen_comprimida, 0, 255).astype(np.uint8))  # Asegura valores
    v lidos
plt.title(f'Imagen Comprimida en Color con k={k}')
plt.axis('off')
plt.show()
```

Se utiliza `Matplotlib` para visualizar la imagen comprimida y `np.clip` asegura que los valores de los píxeles estén dentro del rango válido para imágenes RGB (0 a 255).

1.2.7. Cálculo y Comparación de Tamaños

```
tamano_original = matriz_imagen.size
print(f"Tamano original: {tamano_original}")
print(f"Tamano comprimido: {tamano_comprimido}")
```

Tamano original: 1366200
Tamano comprimido: 422100

`matriz_imagen.size` devuelve el número total de elementos en la imagen original. El tamaño de la imagen original y el tamaño comprimido se imprimen para comparar la eficiencia de la compresión.

1.2.8. Cálculo del Error de Reconstrucción

```
def error_reconstruccion_frobenius(matriz_original, matriz_aproximada):
    error_total = 0
    for i in range(3):  # Iterar sobre cada canal (Red, Green, Blue)
        error_canal =
    np.linalg.norm(matriz_original[:, :, i]-matriz_aproximada[:, :, i], 'fro')
    / np.linalg.norm(matriz_original[:, :, i], 'fro')
        error_total += error_canal
    error_promedio = error_total / 3
    return error_promedio
```

Esta función calcula el error de reconstrucción entre la imagen original y la imagen comprimida utilizando la norma de Frobenius:

La **norma de Frobenius** es una medida de tamaño o magnitud de una matriz. Para una matriz A de tamaño $m \times n$ con elementos a_{ij}, la norma de Frobenius se define como:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$$

En otras palabras, es la raíz cuadrada de la suma de los cuadrados de todos los elementos de la matriz. La norma de Frobenius es una extensión de la norma euclidiana para matrices, y es equivalente a la norma vectorial ℓ_2 aplicada a todos los elementos de la matriz cuando esta se trata como un solo vector.

- Para cada canal (R, G, B), se calcula el error de Frobenius como:

$$\text{error\_canal} = \frac{\|\text{matriz\_original}_{[:,:,i]} - \text{matriz\_aproximada}_{[:,:,i]}\|_F}{\|\text{matriz\_original}_{[:,:,i]}\|_F}$$

- Se calcula el promedio de los errores para los tres canales.

1.2.9. Calcular y Mostrar el Error de Reconstrucción

```
error = error_reconstruccion_frobenius(matriz_imagen, imagen_comprimida)
print(f'Error de reconstrucci n: {error:.4f}')
```

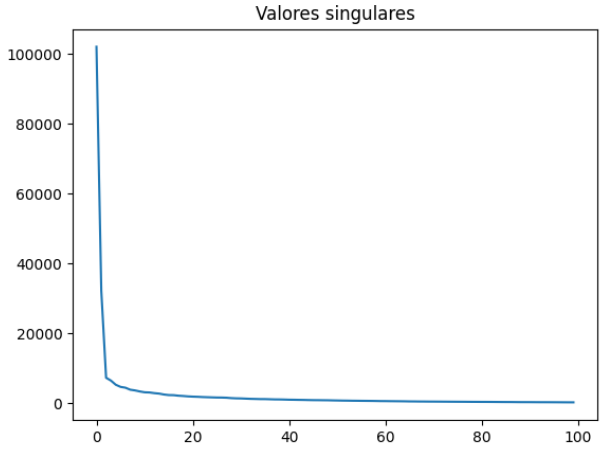
Se calcula el **error de reconstrucción** utilizando la función `error_reconstruccion_frobenius` y se imprime el resultado con 4 decimales.

Error de reconstrucción: 0.0757

1.3. Resultados

Lo interesante de este método es llegar a una solución en el que reduzcamos en gran medida el espacio que ocupa la imagen sin perder mucha calidad en la misma. No deja de ser un problema en el que se busca conseguir una optimización de la memoria manteniendo la información más relevante e importante.

Para encontrar la solución lo primero en lo que nos fijamos es en los valores singulares.



La gráfica representa los **valores singulares** de una matriz ordenados de mayor a menor. Estos valores son útiles para comprender la estructura y la variabilidad de los datos contenidos en la matriz original.

- **Primeros valores singulares:** Los valores singulares más grandes, ubicados en el extremo izquierdo de la gráfica, tienen un valor significativamente mayor en comparación con los siguientes valores. Este comportamiento indica que las primeras componentes capturan la mayor parte de la información contenida en la matriz original. En otras palabras, las primeras componentes son las más representativas y pueden utilizarse para aproximar la estructura principal de los datos.
- **Curva descendente:** La gráfica muestra una rápida caída al inicio, lo que sugiere que pocos valores singulares son suficientes para explicar una gran parte de la estructura de la matriz. Lo cual implica que la matriz original puede ser aproximada de manera eficiente usando solo unas pocas componentes principales.
- **Valores pequeños al final:** A medida que se avanza hacia la derecha en la gráfica, los valores singulares se vuelven cada vez más pequeños y tienden a acercarse a cero. Estos valores pequeños o nulos suelen no aportar información relevante. En el contexto de reducción de dimensionalidad, estos valores pueden omitirse sin pérdida significativa de información.

En nuestro caso los valores son muy cercanos al 0 por eso no aumentamos la k.Tomamos k=100. Ahora vamos a centrarnos en las imágenes.

La imagen original es:

La imagen comprimida es:

Como vemos las imágenes son casi iguales, solo podemos apreciar diferencias si nos fijamos en ciertos detalles haciendo zoom. Además, el tamaño de la imagen comprimida es en torno a un tercio del tamaño de la imagen original, lo cual es una reducción bastante notable.

Por último, calculamos el error de reconstrucción para ver cuan fiable es el método ya que no es suficiente comparando las imágenes a ojo. El error de reconstrucción no deja de ser un error relativo. Aplicamos la siguiente fórmula:

$$\text{error\_canal} = \frac{\|\text{matriz\_original}_{[:,:,i]} - \text{matriz\_aproximada}_{[:,:,i]}\|_F}{\|\text{matriz\_original}_{[:,:,i]}\|_F} \cdot 100$$

El error de reconstrucción nos dá un 7,5 % que, considerando lo que se reduce el tamaño de la imagen, creemos que es aceptable.

Por tanto, teniendo en cuanto la reducción del tamaño y la información que mantenemos, hemos conseguido una solución bastante óptima.