[AwAD 2024] Projekt nr 3.

Maciej Andrzejewski (nr albumu: 333125)
Piotr Kotłowski (nr albumu: 333147)

W tym projekcie skupimy się na zastosowaniu rozkładu macierzy SVD do kompresji obrazów. Przypomnijmy najpierw, czym jest rozkład SVD.

Rozkład SVD

Rozkład SVD to rozkład dowolnej macierzy A o wymiarach $m \times n$ na iloczyn trzech macierzy:

$$A = U\Sigma V^*,$$

gdzie U to ortogonalna macierz $m \times m$, V to ortogonalna macierz $n \times n$, a Σ to macierz diagonalna $m \times n$ zawierająca uszeregowane nierosnąco wartości osobliwe (pierwiastki kwadratowe dodatnich wartości własnych) macierzy A^*A. Ponadto, liczba niezerowych wartości osobliwych z macierzy Σ jest równa rzędowi rozkładanej macierzy. Oznacza to, że wartości osobliwe dostarczają informacji o istotnych składowych macierzy i jej wymiarowości.

Jak znaleźć rozkład SVD macierzy?

Pokażemy to na prostym przykładzie. Niech

$$A = \begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix}.$$

Szukamy macierzy A^*A. Otrzymujemy, że

$$A^*A = \begin{pmatrix} 13 & 12 & 2 \\ 12 & 13 & -2 \\ 2 & -2 & 8 \end{pmatrix}.$$

Jej wartości własne to $\lambda_1 = 25, \lambda_2 = 9, \lambda_3 = 0$, czyli jej wartości osobliwe to

$$\sigma_1 = 5, \quad \sigma_2 = 3.$$

A zatem macierz Σ to

$$\Sigma = \begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix}.$$

Następnie szukamy ortonormalnych wektorów własnych macierzy A^*A, będą to odpowiednio:

$$v_1 = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right)^T, \quad v_2 = \left(\frac{\sqrt{2}}{6}, -\frac{\sqrt{2}}{6}, \frac{2\sqrt{2}}{3}\right)^T, \quad v_3 = \left(-\frac{2}{3}, \frac{2}{3}, \frac{1}{3}\right)^T.$$

A zatem macierz V to

$$V = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{6} & -\frac{2}{3} \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{6} & \frac{2}{3} \\ 0 & \frac{2\sqrt{2}}{3} & \frac{1}{3} \end{pmatrix}.$$

Teraz, szukamy macierzy AA^*:

$$AA^* = \begin{pmatrix} 17 & 8 \\ 8 & 17 \end{pmatrix}.$$

Jej wartości własne to $\lambda_1 = 25, \lambda_2 = 9$, czyli wartości osobliwe to

$$\sigma_1 = 5, \quad \sigma_2 = 3.$$

Jej ortonormalne wektory własne to odpowiednio:

$$u_1 = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right)^T, \quad u_2 = \left(\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}\right)^T.$$

Zatem macierz U to

$$U = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{pmatrix}.$$

Ostatecznie mamy:

$$A = \begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{pmatrix} \begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix} \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{6} & -\frac{2}{3} \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{6} & \frac{2}{3} \\ 0 & \frac{2\sqrt{2}}{3} & \frac{1}{3} \end{pmatrix}^* .$$

Przetwarzanie obrazów

Komputery przechowują obrazy jako odpowiedniej wielkości macierze, której wartości reprezentują nasycenie kolorów w każdym z pikseli. W przypadku obrazów czarnobiałych będzie to po prostu macierz z liczbami z zakresu [0,255] reprezentującymi jasność poszczególnych pikseli (przyjmijmy, że: 0 - zupełnie czarny, 255 - całkowicie biały).

Jak każdą macierz, także opisaną powyżej "macierz obrazu" jesteśmy w stanie rozłożyć przy pomocy rozkładu SVD. Podobnie jak w innych zastosowaniach, pomijając relatywnie małe (w porównaniu do pozostałych) wartości osobliwe w macierzy Σ jesteśmy w stanie zredukować daną macierz do prostszej postaci (a dokładniej zapisać ją jako sumę iloczynów macierzy, które razem zawierają mniej wartości, niż oryginalna macierz) przy utracie małej ilości informacji. Tym sposobem "mało wnoszące" do całości wartości (informacje) są pomijane, zdjęcie nie różni się znacząco ostrością od oryginału, natomiast zajmuje mniej pamięci na dysku.

Korzystając z języka Python, bazując głównie na podstawowych funkcjach jego bibliotek matematycznych, stworzyliśmy program, który przetwarza obrazy w wyżej opisany sposób.

Zobaczymy jego zastosowanie na przykładzie. Do tego celu użyjemy zdjęcia, w którego macierzowej reprezentacji rozłożonej za pomocą rozkładu SVD będziemy pomijać coraz to więcej wartości osobliwych i patrzeć, jak wpływa to na jakość obrazu.

Oryginalny obraz:

Rysunek 1: Oryginalny obraz (512 wartości osobliwych)

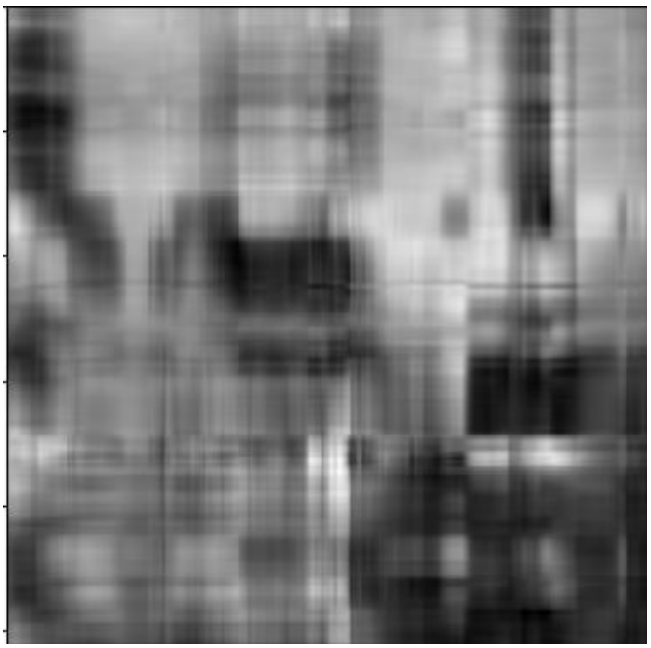
Obrazy ze zredukowaną liczbą wartości własnych:

Rysunek 2: Obraz po pozostawieniu pierwszych 300 wartości osobliwych

Rysunek 3: Obraz po pozostawieniu pierwszych 100 wartości osobliwych

Rysunek 4: Obraz po pozostawieniu pierwszych 60 wartości osobliwych

Rysunek 5: Obraz po pozostawieniu pierwszych 20 wartości osobliwych



Rysunek 6: Obraz po pozostawieniu pierwszych 5 wartości osobliwych

W zależności od tego, jak niska rozdzielczość zdjęcia będzie wystarczająca dla naszych potrzeb, tak dużo wartości osobliwych będziemy mogli usunąć. Zauważmy, że jakoś zdjęcia znacząco spadła dopiero gdy pozostało mniej niż 100 wartości osobliwych (z pierwotnych 512). Pozwala nam to wnioskować, że powyższym sposobem, pozostawiając minimalną niezbędną liczbę wartości osobliwych, jesteśmy w stanie zmniejszyć ilość pamięci, jakiej potrzebujemy, aby dany obraz przechowywać, zachowując jednocześnie jego dobrą jakość.

Kod napisanego programu:

```
import cmath
from matplotlib.image import imread
import copy
import math
import numpy as np
import numpy.linalg as nplg


def Scalar(vector1, vector2):
    suma = 0
    for i in range(len(vector1)):
        suma += vector1[i] * vector2[i]
    return suma


def Ortogonalizacja(Matrix):
    toOrthogonal = copy.deepcopy(Matrix)
    toOrthogonal[0] = np.array(toOrthogonal[0])
    vectortoSubstract = []
    for i in range(1, len(toOrthogonal)):
        vector = np.array(toOrthogonal[i])
        VectorsToSub = [0] * len(toOrthogonal[i])
        for k in range(i):

            scalar = Scalar(vector, toOrthogonal[k]) / Scalar(toOrthogonal[k], toOrthogonal[k]
                                                              )

            for s in range(len(toOrthogonal[k])):
                vectortoSubstract.append(toOrthogonal[k][s] * scalar)
            VectorsToSub = np.add(VectorsToSub, vectortoSubstract)
            vectortoSubstract = []
        toOrthogonal[i] = np.subtract(toOrthogonal[i], VectorsToSub)
```

```
    return toOrthogonal;


def DoubleVEctors(EigenValues, EigenVectors):
    previous = float("inf")
    toOrthogonal = []
    Result = []
    transposedVectors = np.transpose(copy.deepcopy(EigenVectors))
    for k in range(len(EigenValues)):
        if previous != EigenValues[k]:
            Occur = np.count_nonzero(EigenValues == EigenValues[k])
            if Occur != 1:
                for s in range(Occur):
                    toOrthogonal.append(transposedVectors[s])
                Result = Ortogonalizacja(toOrthogonal)
            else:
                Result.append(transposedVectors[k])
        previous = EigenValues[k]
    return np.transpose(Result)


def GiveMatrix(input):
    matrix = np.matmul(np.transpose(input), input)
    return matrix


def ScientificToFloatVector(vector):
    for i in range(len(vector)):
        # vector[i]=float(vector[i]) #Problem tutaj z konwertowaniem
        vector[i] = '{:f}'.format(vector[i])
    return vector


def EigenToSingular(matrix):
    matrixResult = [0] * len(matrix)
    for i in range(len(matrix)):
        matrixResult[i] = cmath.sqrt(matrix[i])
    return matrixResult


def Norm(vector):
    ResultNorm = 0
    for k in range(len(vector)):
        ResultNorm += vector[k] * vector[k]
    return cmath.sqrt(ResultNorm)


def NormForEigenVectors(matrix):
    # print(matrix)
    matrixOperation = np.transpose(copy.deepcopy(matrix))
    matrixResult = copy.deepcopy(matrixOperation)
    # print(matrixOperation)
    for i in range(len(matrixOperation)):
        norm = Norm(matrixOperation[i])
        for j in range(len(matrixOperation[i])):
            matrixResult[i][j] = matrixOperation[i][j] / norm
    matrixResult = np.transpose(matrixResult)
    return matrixResult


def matrixU(singularvalues, A, eigenvectors):
    matrixResult = []
    RightFormatEvectors = np.transpose(copy.deepcopy(eigenvectors))
    for k in range(len(A)):
        matrixResult.append(np.array(np.matmul(A, RightFormatEvectors[k])) / singularvalues[k]
                                                                           )
    return np.transpose(matrixResult)


def SingValuestoMatrix(input, singularvalues):
    singularvaluesSorted = copy.deepcopy(singularvalues)
    singularvaluesSorted = np.flip(np.sort(singularvaluesSorted))
    Result = np.zeros((len(input), len(input[0])))
    for k in range(len(input)):
        Result[k][k] = singularvaluesSorted[k]
    return Result
```

```
def RightOrder(values, matrix):
    sort_index = np.flip(np.argsort(values))

    matrixResult = copy.deepcopy(matrix)
    return matrixResult[:, sort_index]


def ReverseMatrix(Matrixinput):
    Matrix = copy.deepcopy(Matrixinput)
    for i in range(len(Matrix)):
        Matrix[i][i] = 1 / Matrix[i][i]
    return np.transpose(Matrix)


def Noises(List, k):
    for i in range(k, len(List), 1):
        List[i] = 0

    return List


def main():
    a = imread("skimage_astronaut.png")

    matrix = GiveMatrix(a)
    EigenValues = np.array(nplg.eig(matrix)[0])
    EigenVectors = np.array(nplg.eig(matrix)[1])

    # Liczenie singular values
    # EigenValues=ScientificToFloatVector(EigenValues)

    SingularValues = EigenToSingular(EigenValues)

    # SingularValues=np.flip(np.sort(SingularValues))
    #MatrixSingValus = SingValuestoMatrix(a, SingularValues)

    # Liczenie v transponowanego

    EigenVectors = RightOrder(SingularValues, EigenVectors)
    EigenValues = np.flip(np.sort(EigenValues))
    EigenVectors = DoubleVEctors(EigenValues, EigenVectors)
    EigenVectors = NormForEigenVectors(EigenVectors)

    # Liczenie U
    SingularValuesforU = np.flip(np.sort(SingularValues))
    U = (matrixU(SingularValuesforU, a, EigenVectors))

    # print(U)
    # print(MatrixSingValus)
    # print(np.transpose(EigenVectors))

    # Noises
    k=int(input("Ile chcesz zostawic wartosci osobliwych?"))
    MatrixSingValusNoise = SingValuestoMatrix(a, Noises(SingularValues, k)) #Zmienna k

    firstMultiNoise = np.matmul(U, MatrixSingValusNoise)
    resultNoise = np.matmul(firstMultiNoise, np.transpose(EigenVectors))

   # firstMulti = np.matmul(U, MatrixSingValus)
   # result = np.matmul(firstMulti, np.transpose(EigenVectors))

    from matplotlib import pyplot as plt

    plt.imshow(resultNoise.real, interpolation='nearest',cmap="gray")
    #plt.imshow(result.real,interpolation='nearest',cmap="gray")
    plt.show()
```