

Algorytmy macierzowe

Laboratorium 3 - Hierarchiczna kompresja macierzy

Kacper Wąchała, Szymon Hołysz

1. Zastosowane algorytmy

1.1. Metoda potęgowa

Metoda potęgowa to algorytm iteracyjny pozwalający wyznaczyć największą wartość własną oraz odpowiadający jej wektor własny danej macierzy M . Polega ona na kolejnych mnożeniacach wektora początkowego przez macierz M oraz normalizacji wektora w każdej iteracji. Wynikiem algorytmu jest przybliżenie wektora własnego macierzy M odpowiadającego największej wartościowej, z którego można wyliczyć jej wartość wykorzystując iloraz Rayleigh'a.

Algorithm 1: Metoda potęgowa

```
1: function POWER_ITERATION( $M$ ,  $max\_iter$ ,  $eps$ )
2:    $m, n \leftarrow \text{SHAPE}(M)$ 
3:    $v \leftarrow \text{RANDOMVECTOR}(n)$ 
4:    $v \leftarrow \frac{v}{\|v\|}$ 
5:    $prev \leftarrow \text{EMPTYVECTOR}(n)$ 
6:   for  $i < max\_iter$  do
7:      $prev \leftarrow v$ 
8:      $v \leftarrow M \cdot v$ 
9:     if  $\|v\| < eps$  then
10:       return  $v, 0$ 
11:     end
12:      $v \leftarrow \frac{v}{\|v\|}$ 
13:     if  $\forall i |v_i - prev_i| < eps$  then
14:       BREAK
15:     end
16:   end
17:    $\lambda \leftarrow \frac{v^T M v}{v^T v}$                                  $\triangleright$  Iloraz Rayleigh'a
18:   return  $v, \lambda$ 
19: end
```

1.2. Częściowe SVD

Algorytm SVD polega na dekompozycji danej macierzy $M \in \mathbb{R}^{m \times n}$ na trzy macierze U , D i V , takie że $M = UDV^T$, gdzie $U \in \mathbb{R}^{m \times m}$, $V \in \mathbb{R}^{n \times n}$ to macierze unitarne, a $D \in \mathbb{R}^{m \times n}$ to macierz diagonalna $D_{1,1} \geq D_{2,2} \geq \dots \geq D_{k,k} \geq D_{k+1,k+1} = \dots = D_{\min(m,n)}$, $D_{\min(m,n)} = 0$. Wartości na przekątnej macierzy diagonalnej D to wartości osobliwe macierzy M . Algorytm częściowego SVD do dekompozycji macierzy M wykorzystuje jedynie k jej największych wartości osobliwych.

Algorithm 2: Częściowe SVD

```
1: function TRUNCATED_SVD( $M, r, \varepsilon$ )
2:    $m, n \leftarrow \text{SHAPE}(M)$ 
3:    $U \leftarrow \text{ZEROS}(m, r)$ 
4:    $D \leftarrow \text{ZEROS}(r)$ 
5:    $V \leftarrow \text{ZEROS}(n, r)$ 
6:    $A \leftarrow M^T M$ 
7:   for  $rank < r$  do
8:      $v, \lambda \leftarrow \text{POWER_ITERATION}(A)$ 
9:     if  $\lambda < \varepsilon$  then
10:      BREAK
11:    end
12:     $\sigma \leftarrow \sqrt{\lambda}$ 
13:     $D[rank] \leftarrow \sigma$ 
14:     $V[:, rank] \leftarrow v$ 
15:     $U[:, rank] \leftarrow \frac{M * v}{\sigma}$ 
16:     $A \leftarrow A - \lambda * v \otimes v$ 
17:  end
18:  return  $U, D, V^T$ 
19: end
```

Poniżej znajdują się funkcje pomocnicze pozwalające wykorzystać SVD do kompresji bitmap:

1.3. Kompresja macierzy

Algorithm 3: Kompresja macierzy

```
1: function COMPRESS_MATRIX( $A, U, D, V, r, \varepsilon$ )
2:    $significant\_sigma \leftarrow D[D > \varepsilon]$ 
3:    $rank \leftarrow \min(r, significant\_sigma)$ 
4:   return NODE( $rank, A.shape, U=U[:, :rank], D=D[:rank], V=V[:rank, :]$ )
5: end
```

1.4. Budowa drzewa

Algorithm 4: Budowa drzewa

```
1: function CREATE_TREE( $A, rank, \varepsilon, min\_size$ )
2:    $m, n \leftarrow \text{SHAPE}(A)$ 
3:    $U, D, V \leftarrow \text{TRUNCATED\_SVD}(A, rank + 1, \varepsilon)$ 
4:   if  $\min(A.shape) \leq min\_size \vee D[rank] \leq \varepsilon$  then
5:      $root \leftarrow \text{COMPRESS\_MATRIX}(A, U, D, V, rank, \varepsilon)$ 
6:   else
7:      $root \leftarrow \text{NODE}(0, A.shape)$ 
8:      $\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \leftarrow A$ 
9:     for  $A_i$  do
10:      if  $A_i.size > 0$  then
11:         $root.children.append(\text{Create\_Tree}(A_i, rank, \varepsilon, min\_size))$ 
12:      end
13:    end
14:  end
15:  return  $result$ 
16: end
```

1.5. Rekonstrukcja macierzy

Algorithm 5: Rekonstrukcja macierzy

```
1: function REBUILD_MATRIX( $node$ )
2:   if  $node.children = []$  then
3:     return  $node.U * \text{DIAG}(node.D) * node.V$ 
4:   end
5:    $A_1 \leftarrow \text{REBUILD\_MATRIX}(node.children[0])$ 
6:    $A_2 \leftarrow \text{REBUILD\_MATRIX}(node.children[1])$ 
7:    $A_3 \leftarrow \text{REBUILD\_MATRIX}(node.children[2])$ 
8:    $A_4 \leftarrow \text{REBUILD\_MATRIX}(node.children[3])$ 
9:   return  $\begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$ 
10: end
```

2. Implementacja

2.1. Metoda potęgowa

```
def power_iteration(M, max_iter=1000, eps=1e-8):
    v = np.random.rand(M.shape[1])
    v /= np.linalg.norm(v)
    prev = np.empty(M.shape[1])
    for _ in range(max_iter):
        prev[:] = v
        v = np.dot(M, v)
        norm = np.linalg.norm(v)
        if norm < eps:
            return v, 0
        v /= norm
        if np.allclose(v, prev, atol=eps):
            break
    eigval = np.dot(v, np.dot(M, v)) / np.dot(v, v)
    return v, eigval
```

2.2. Częściowe SVD

```
def truncated_svd(M, r, eps):
    M = M.astype(np.float64)
    U = np.zeros([M.shape[0], r])
    D = np.zeros(r)
    V = np.zeros([M.shape[1], r])
    A = M.T @ M
    for rank in range(r):
        eigvec, eigval = power_iteration(A)
        if eigval < eps:
            break
        singular = np.sqrt(eigval)
        D[rank] = singular
        V[:, rank] = eigvec
        U[:, rank] = M @ eigvec / singular
        A -= eigval * np.outer(eigvec, eigvec)
    return U, D, V.T
```

2.3. Kompresja macierzy

```
def CompressMatrix(A, U, D, V, r, eps):
    significant = D > eps
    rank = min(r, significant.sum())
    U_c = U[:, :rank]
    D_c = D[:rank]
    V_c = V[:rank, :]
    return Node(rank, A.shape, U=U_c, V=V_c, D=D_c)
```

2.4. Budowa drzewa

```
def create_tree(A, rank, eps=1e-10, min_size=2):
    U, D, V = truncated_svd(A, rank + 1, eps)

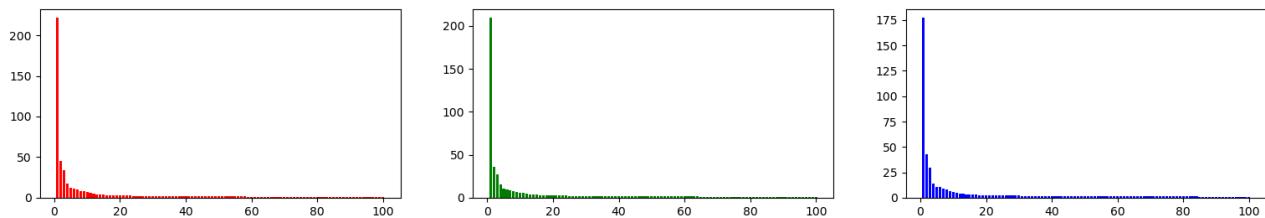
    if min(A.shape) ≤ min_size or D[rank] ≤ eps:
        root = CompressMatrix(A, U, D, V, rank, eps)
    else:
        root = Node(0, A.shape)
        mid_row = A.shape[0] // 2
        mid_col = A.shape[1] // 2

        submatrices = [
            A[:mid_row, :mid_col],
            A[:mid_row, mid_col:],
            A[mid_row:, :mid_col],
            A[mid_row:, mid_col:]
        ]

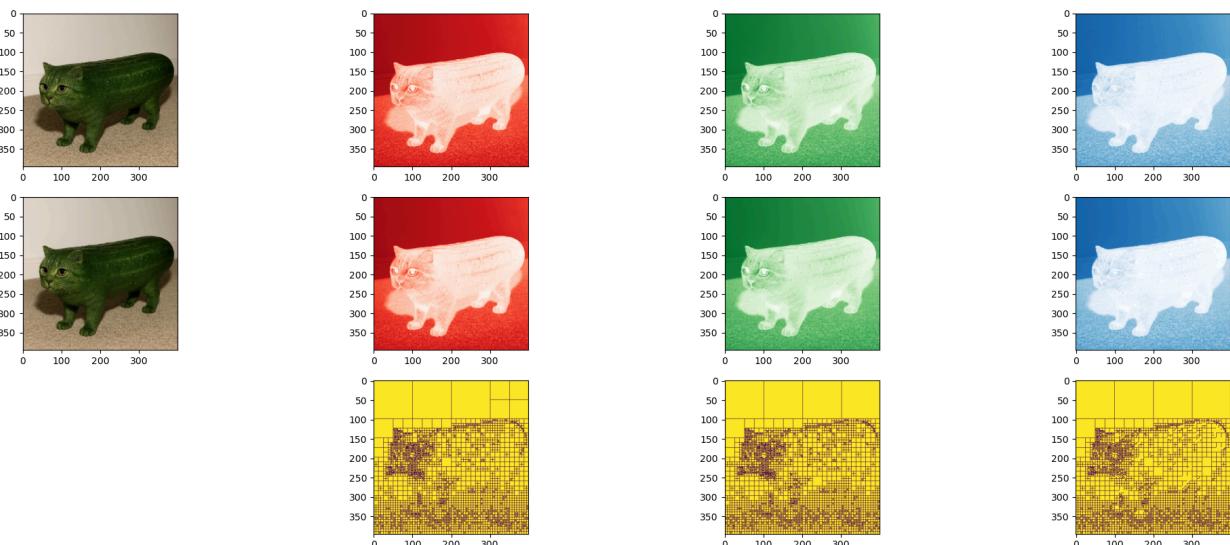
        for subm in submatrices:
            if subm.size > 0:
                root.children.append(create_tree(subm, rank, eps, min_size))

    return root
```

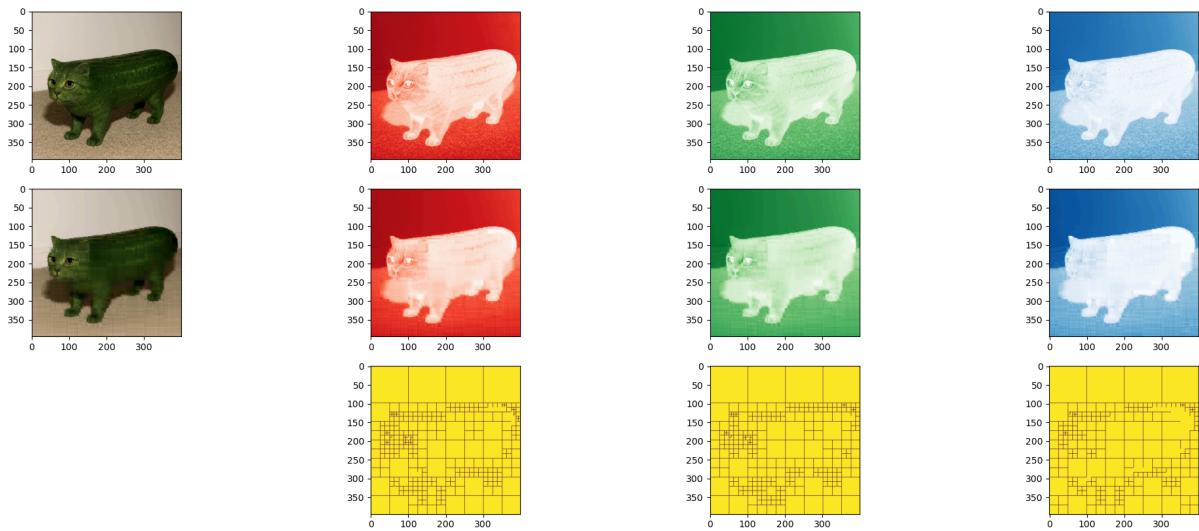
3. Wyniki



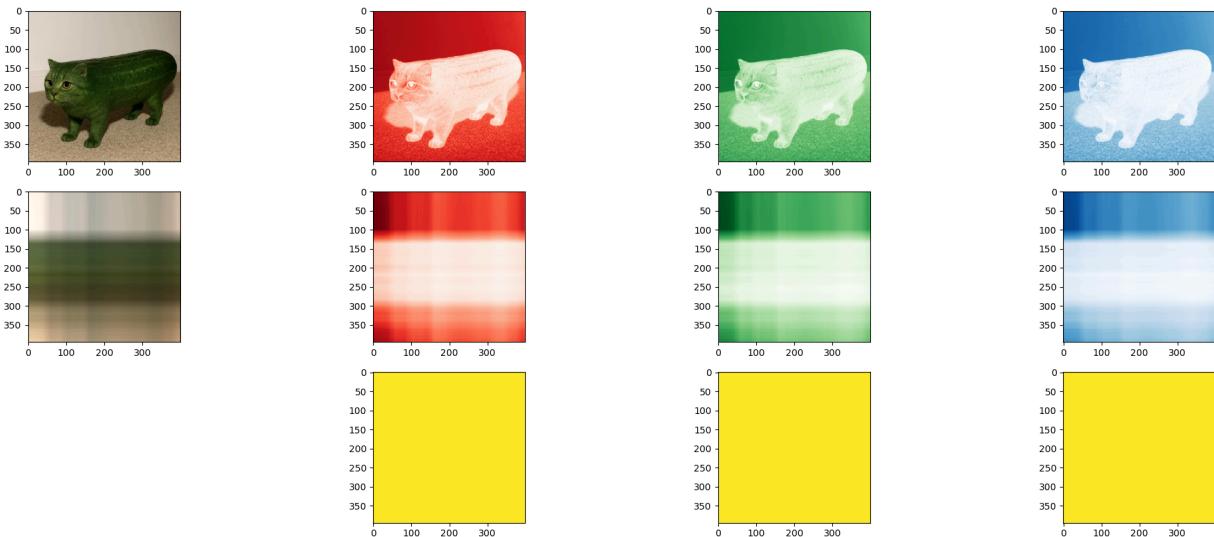
Rysunek 1: Histogram wartości osobliwych w zależności od rzędu, rank_max = 100



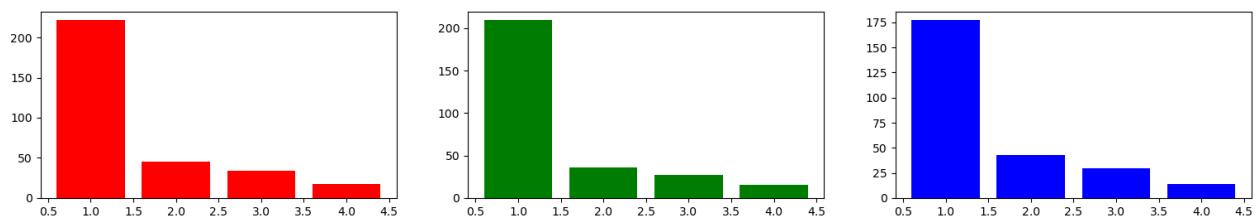
Rysunek 2: Porównanie macierzy kompresji i wynikowej bitmapy RGB do obrazu oryginalnego oraz graficzna reprezentacja drzewa (rank = 1, eps = 0.05)



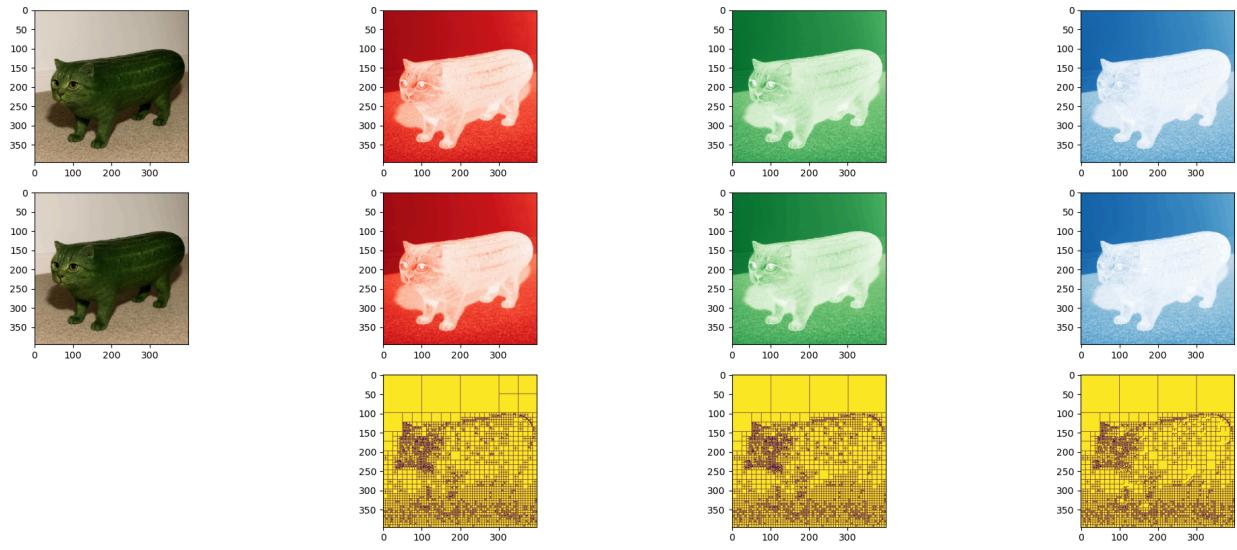
Rysunek 3: Porównanie macierzy kompresji i wynikowej bitmapy RGB do obrazu oryginalnego oraz graficzna reprezentacja drzewa ($\text{rank} = 1$, $\text{eps} = 1$)



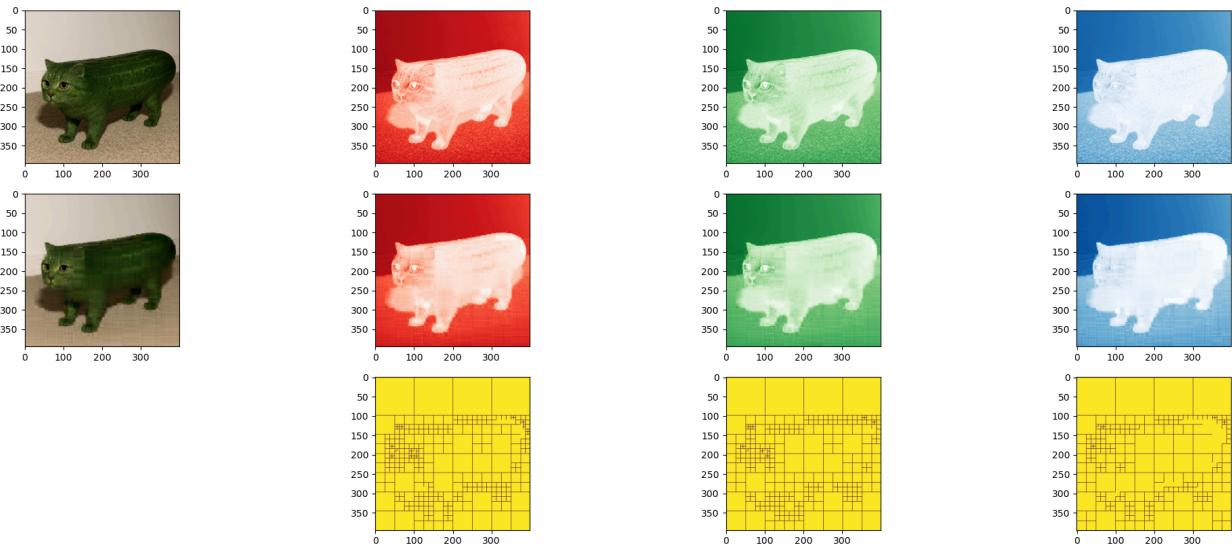
Rysunek 4: Porównanie macierzy kompresji i wynikowej bitmapy RGB do obrazu oryginalnego oraz graficzna reprezentacja drzewa ($\text{rank} = 1$, $\text{eps} = 60$)



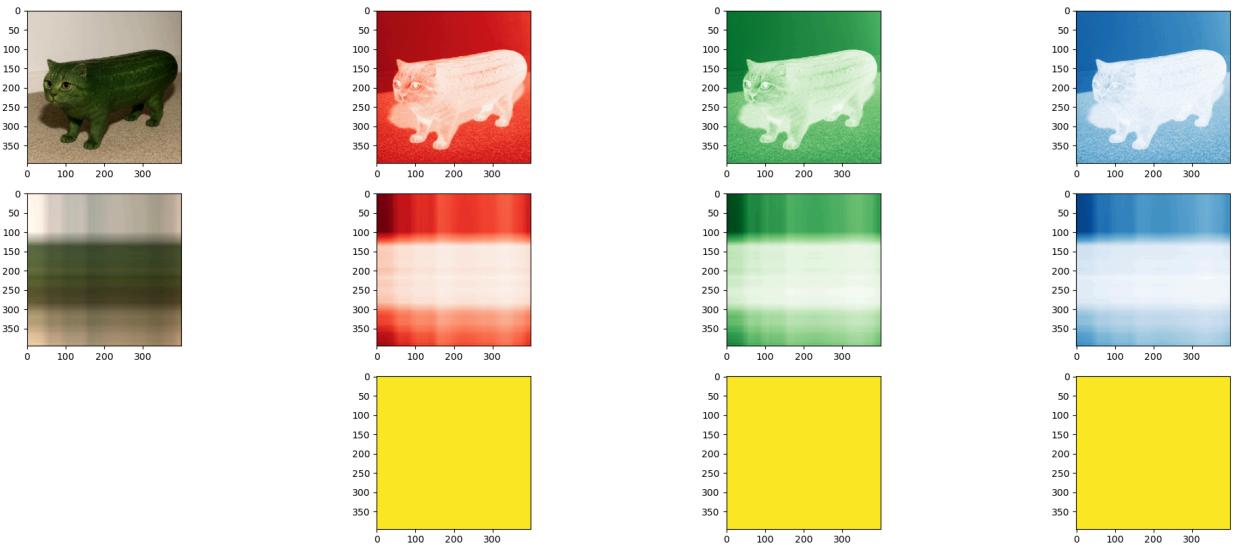
Rysunek 5: Histogram wartości osobliwych w zależności od rzędu, $\text{rank_max} = 4$



Rysunek 6: Porównanie macierzy kompresji i wynikowej bitmapy RGB do obrazu oryginalnego oraz graficzna reprezentacja drzewa ($\text{rank} = 4$, $\text{eps} = 0.05$)



Rysunek 7: Porównanie macierzy kompresji i wynikowej bitmapy RGB do obrazu oryginalnego oraz graficzna reprezentacja drzewa ($\text{rank} = 1$, $\text{eps} = 1$)



Rysunek 8: Porównanie macierzy kompresji i wynikowej bitmapy RGB do obrazu oryginalnego oraz graficzna reprezentacja drzewa ($\text{rank} = 1$, $\text{eps} = 60$)

4. Wnioski

Kompresja najlepiej radzi sobie dla `max_rank = 4` i `eps = 1` (Mniej więcej równiejskie wartości osobiściej). Przy tych parametrach obrazek nadal jest dość wyraźny, a zajmuje mniej miejsca.