Q1

June 17, 2020

[1]:
```python
import numpy as np
import tensorly as tl
from tensorly.decomposition import tucker

from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))
```

```
<IPython.core.display.HTML object>
```

Question 1. Tensor Decomposition Reconstructions (15 points)

Part 1. Kruskal tensors are a way of representing tensor decompositions as a weighted sum of outer products.

$\chi = \sum_r \lambda_r U_{1r} \circ U_{2r} \circ \ldots \circ U_{nr}$ for each rank of the decomposition, r, and rank of the original tensor, n.

a) Given the following rank-2 CP decomposition:

$$\lambda = \begin{pmatrix} 39.288 & 10.676 \end{pmatrix} U_1 = \begin{pmatrix} 0.5719 & 0.1469 \\ 0.5885 & 0.9817 \\ 0.5715 & -0.1210 \end{pmatrix} U_2 = \begin{pmatrix} 0.5121 & -0.4042 \\ 0.6284 & 0.5877 \\ 0.5856 & 0.7009 \end{pmatrix}$$

$$U_3 = \begin{pmatrix} 0.5605 & -0.3179 \\ 0.4921 & -0.3682 \\ 0.6661 & 0.8737 \end{pmatrix} U_4 = \begin{pmatrix} 0.7502 & -0.9201 \\ 0.6612 & 0.3917 \end{pmatrix}$$

Write out the calculation of the first outer product $U_{1,1} \circ U_{2,1}$

Answer:

$U_{1,1} \circ U_{2,1} = \mathrm{U}_{1,1} * \mathrm{U}_{2,1}{}^{\mathrm{T}}$

Where $\mathrm{U}_{1,1} = \begin{pmatrix} 0.5719 \\ 0.5885 \\ 0.5715 \end{pmatrix}$ and $\mathrm{U}_{2,1}{}^{\mathrm{T}} = \begin{pmatrix} 0.5121 & 0.6284 & 0.5856 \end{pmatrix}$

$$\begin{pmatrix} (0.5719 * 0.5121) & (0.5719 * 0.6284) & (0.5719 * 0.5856) \\ (0.5885 * 0.5121) & (0.5885 * 0.6284) & (0.5885 * 0.5856) \\ (0.5715 * 0.5121) & (0.5715 * 0.6284) & (0.5715 * 0.5856) \end{pmatrix} = \boxed{\begin{pmatrix} 0.2929 & 0.3594 & 0.3349 \\ 0.3014 & 0.3698 & 0.3446 \\ 0.2927 & 0.3591 & 0.3347 \end{pmatrix}}$$

0.0.1 Confirm written solution programmatically

[2]:
```
u11 = np.array([0.5719, 0.5885, 0.5715])
u21 = np.array([0.5121, 0.6284, 0.5856])
np.outer(u11, u21)
```

[2]:
```
array([[0.29286999, 0.35938196, 0.33490464],
       [0.30137085, 0.3698134 , 0.3446256 ],
       [0.29266515, 0.3591306 , 0.3346704 ]])
```

0.1 b) Either by hand or in code, calculate:

0.1.1 $\lambda_1 \; U_{1,1} \circ U_{2,1} \circ U_{3,1} \circ U_{4,1}$

[3]:
```
lam = np.array([39.288 , 10.676])

u1 = np.array([[0.5719, 0.1469], [0.5885,0.9817], [0.5715, -0.1210]])

u2 = np.array([[0.5121, -0.4042], [0.6284,0.5877], [0.5856, 0.7009]])

u3 = np.array([[0.5605, -0.3179], [0.4921,-0.3682], [0.6661, 0.8737]])

u4 = np.array([[0.7502, -0.9201], [0.6612,0.3917]])
```

[4]:
```
#intialize empty placeholder for outer product of the first rank
prod1 = np.zeros((3,3,3,2))

#extract first column from each U matrix
u11 = u1[:,0]
u21 = u2[:,0]
```

```
u31 = u3[:,0]
u41 = u4[:,0]

#perform outer product
for i in range(len(u41)):               #length should be 2 since there's 2␣
↪values in the columns of U4
    for j in range(len(u31)):           #length should be 3 since there's 3␣
↪values in the columns of U3
        for k in range(len(u21)):       #length should be 3 since there's 3␣
↪values in the columns of U2
            for l in range(len(u11)): #length should be 3 since there's 3␣
↪values in the columns of U1

                prod1[l][k][j][i] = u11[l] * u21[k] * u31[j] * u41[i]

#multiply outer product by the first lambda value
prod1 = prod1 * lam[0]
prod1
```

[4]:
```
array([[[[4.8382407 , 4.26425586],
         [4.24781132, 3.7438721 ],
         [5.74978078, 5.06765536]],

        [[5.93702491, 5.23268577],
         [5.21250661, 4.59412073],
         [7.05557946, 6.21854058]],

        [[5.5326572 , 4.87629024],
         [4.85748547, 4.28121754],
         [6.57502758, 5.79499899]]],


       [[[4.97867573, 4.38803038],
         [4.37110852, 3.85254193],
         [5.91667423, 5.2147494 ]],

        [[6.10935331, 5.38456999],
         [5.36380511, 4.72746992],
         [7.26037509, 6.39904027]],

        [[5.6932484 , 5.01782971],
         [4.99847911, 4.40548439],
         [6.76587469, 5.96320494]]],


       [[[4.83485672, 4.26127335],
         [4.24484031, 3.74125355],
```

```
          [5.74575925, 5.06411093]],

         [[5.93287241, 5.22902591],
          [5.20886087, 4.5909075 ],
          [7.05064463, 6.21419119]],

         [[5.52878753, 4.87287965],
          [4.85408804, 4.27822315],
          [6.57042886, 5.79094583]]]])
```

0.1.2 $\lambda_1\ U_{1,2} \circ U_{2,2} \circ U_{3,2} \circ U_{4,2}$

[5]:
```
#initialize empty placeholder for outer product fo the second rank
prod2 = np.zeros((3,3,3,2))

#extract second column from each U matrix
u12 = u1[:,1]
u22 = u2[:,1]
u32 = u3[:,1]
u42 = u4[:,1]

#perform outer product
for i in range(len(u42)):                 #length should be 2 since there's 2␣
↪values in the columns of U4
    for j in range(len(u32)):             #length should be 3 since there's 3␣
↪values in the columns of U3
        for k in range(len(u22)):         #length should be 3 since there's 3␣
↪values in the columns of U2
            for l in range(len(u12)): #length should be 3 since there's 3␣
↪values in the columns of U1

                prod2[l][k][j][i] = u12[l] * u22[k] * u32[j] * u42[i]

#multiply outer product by the second lambda value
prod2 = prod2 * lam[1]
prod2
```

[5]:
```
array([[[[-0.18541814,  0.07893521],
         [-0.21475609,  0.0914248 ],
         [ 0.50959368, -0.21694147]],

        [[ 0.26959486, -0.11477047],
         [ 0.31225174, -0.13293012],
         [-0.74094064,  0.31542925]],
```

```
 [[ 0.32152295, -0.13687701],
  [ 0.3723962 , -0.1585345 ],
  [-0.88365713,  0.37618574]]],


[[[-1.23910818,  0.52750644],
  [-1.43516713,  0.6109716 ],
  [ 3.40550115, -1.44977155]],

 [[ 1.80164245, -0.76698549],
  [ 2.08670887, -0.88834242],
  [-4.95154139,  2.10794344]],

 [[ 2.14866631, -0.91471861],
  [ 2.48864088, -1.05945075],
  [-5.90528391,  2.51396556]]],


[[[ 0.15272699, -0.06501811],
  [ 0.17689235, -0.07530566],
  [-0.41974701,  0.17869243]],

 [[-0.22206248,  0.09453524],
  [-0.25719851,  0.10949316],
  [ 0.61030509, -0.25981579]],

 [[-0.26483511,  0.11274417],
  [-0.30673887,  0.13058321],
  [ 0.72785918, -0.30986028]]]]])
```

0.1.3 χ the full reconstruction

[6]:
```
X = prod1 + prod2
X.shape
```

[6]:
```
(3, 3, 3, 2)
```

[7]:
```
X
```

[7]:
```
array([[[[4.65282255, 4.34319107],
         [4.03305524, 3.8352969 ],
         [6.25937447, 4.85071389]],

        [[6.20661977, 5.11791531],
         [5.52475835, 4.46119061],
         [6.31463882, 6.53396982]],
```

```
 [[5.85418015, 4.73941323],
  [5.22988167, 4.12268304],
  [5.69137045, 6.17118473]]],


[[[3.73956755, 4.91553682],
  [2.93594139, 4.46351353],
  [9.32217538, 3.76497785]],

 [[7.91099576, 4.6175845 ],
  [7.45051398, 3.8391275 ],
  [2.30883371, 8.50698371]],

 [[7.84191472, 4.10311109],
  [7.48711999, 3.34603364],
  [0.86059077, 8.47717049]]],


[[[4.98758371, 4.19625523],
  [4.42173266, 3.66594789],
  [5.32601224, 5.24280336]],

 [[5.71080993, 5.32356115],
  [4.95166236, 4.70040065],
  [7.66094972, 5.9543754 ]],

 [[5.26395243, 4.98562382],
  [4.54734917, 4.40880637],
  [7.29828804, 5.48108555]]]])
```

0.1.4 Use tensorly and compare results of manual calculation from library implementation

[8]:
```
X_tl = tl.kruskal_to_tensor((lam, [u1,u2,u3,u4]))
X_tl
```

[8]:
```
array([[[[4.65282255, 4.34319107],
         [4.03305524, 3.8352969 ],
         [6.25937447, 4.85071389]],

        [[6.20661977, 5.11791531],
         [5.52475835, 4.46119061],
         [6.31463882, 6.53396982]],

        [[5.85418015, 4.73941323],
```

```
  [5.22988167, 4.12268304],
  [5.69137045, 6.17118473]]],


[[[3.73956755, 4.91553682],
  [2.93594139, 4.46351353],
  [9.32217538, 3.76497785]],

 [[7.91099576, 4.6175845 ],
  [7.45051398, 3.8391275 ],
  [2.30883371, 8.50698371]],

 [[7.84191472, 4.10311109],
  [7.48711999, 3.34603364],
  [0.86059077, 8.47717049]]],


[[[4.98758371, 4.19625523],
  [4.42173266, 3.66594789],
  [5.32601224, 5.24280336]],

 [[5.71080993, 5.32356115],
  [4.95166236, 4.70040065],
  [7.66094972, 5.9543754 ]],

 [[5.26395243, 4.98562382],
  [4.54734917, 4.40880637],
  [7.29828804, 5.48108555]]]])
```

Part 2. A Tucker decomposition of the same original tensor is:

$$G_{1,1}=\begin{pmatrix}38.946 & 0.8653\\ 0.9666 & -4.8832\end{pmatrix} \quad G_{2,1}=\begin{pmatrix}-0.4799 & -0.0792\\ -1.7302 & -4.3675\end{pmatrix}$$

$$G_{1,2}=\begin{pmatrix}0.7059 & -1.6496\\ 0.7553 & -1.1648\end{pmatrix} \quad G_{2,2}=\begin{pmatrix}5.7493 & -3.3204\\ -2.0019 & 7.6587\end{pmatrix}$$

$$U_1=\begin{pmatrix}0.5661 & -0.1945\\ 0.6005 & -0.5685\\ 0.5648 & 0.7994\end{pmatrix} \quad U_2=\begin{pmatrix}0.5031 & 0.8331\\ 0.6345 & -0.1755\\ 0.5867 & -0.5246\end{pmatrix} \quad U_3=\begin{pmatrix}0.5773 & -0.3364\\ 0.5013 & -0.5733\\ 0.6445 & 0.7471\end{pmatrix}$$

$$U_4=\begin{pmatrix}0.7524 & -0.6587\\ 0.6587 & 0.7524\end{pmatrix}$$

Compute the reconstruction of the Tucker decomposition.

[9]:
```
g11 = np.array([[38.946, 0.8653],
                [0.9666, -4.8832]])

g12 = np.array([[0.7059, -1.6496],
                [0.7553, -1.1648]])

g21 = np.array([[-0.4799, -0.0792],
                [-1.7302, -4.3675]])

g22 = np.array([[5.7493, -3.3204],
                [-2.0019, 7.6587]])

g = np.zeros((2,2,2,2))


g[:,:,0,0] = g11
g[:,:,1,0] = g12
g[:,:,0,1] = g21
g[:,:,1,1] = g22

g = tl.tensor(g, dtype=tl.float32)
```

[10]:
```
u1 = np.array([[0.5661, -0.1945],
               [0.6005, -0.5685],
               [0.5648, 0.7994]])

u2 = np.array([[0.5031, 0.8331],
```

```
               [0.6345, -0.1755],
               [0.5867, -0.5246]])

u3 = np.array([[0.5773, -0.3364],
               [0.5013, -0.5733],
               [0.6445, 0.7471]])

u4 = np.array([[0.7524, -0.6587],
               [0.6587, 0.7524]])
```

0.1.5 Manual Calculation

[11]:
```
prod = []
for p in range(2):
    for q in range(2):
        for r in range(2):
            for s in range(2):

                prod.append(np.outer(np.outer(np.outer((g[p][q][r][s] * u1[:
 ↪,p]), u2[:,q]), u3[:,r]) , u4[:,s]))
```

[12]:
```
result = np.zeros(prod[0].shape)

for i in prod:
    result += i

result = result.reshape((3,3,3,2))
result
```

[12]:
```
array([[[[ 4.93169791,  5.29221616],
         [ 4.1998829 ,  4.88708718],
         [ 5.83543511,  4.74410601]],

        [[ 6.52414397,  4.33674962],
         [ 6.14284537,  3.09887053],
         [ 5.37677372,  7.50444614]],

        [[ 6.1225886 ,  3.3192871 ],
         [ 5.93006553,  1.95986379],
         [ 4.38583683,  7.388613  ]]],


       [[[ 4.68851114,  7.26166459],
         [ 3.57800094,  6.98710315],
         [ 7.20373333,  5.38633794]],
```

```
        [[ 6.91614031,  4.26511325],
         [ 6.74001646,  2.84167726],
         [ 4.7891852 ,  8.20297477]],

        [[ 6.69150931,  2.49794405],
         [ 6.89045511,  0.863456  ],
         [ 3.15897946,  8.00158859]]],


       [[[ 6.40945058,  0.76002819],
         [ 6.5960878 , -0.0698415 ],
         [ 3.04148456,  3.76233945]],

        [[ 6.52137912,  5.24614314],
         [ 5.51461142,  4.31375332],
         [ 7.87237581,  6.82201766]],

        [[ 5.56851484,  6.11785216],
         [ 4.27055427,  5.28945591],
         [ 8.47204259,  6.92181322]]]])
```

0.1.6 Tensorly Calculation

[13]:
```
result_tl = tl.tucker_to_tensor((g, [u1,u2,u3,u4]))
```

[14]:
```
result_tl
```

[14]:
```
array([[[[ 4.93169791,  5.29221616],
         [ 4.1998829 ,  4.88708718],
         [ 5.83543511,  4.74410601]],

        [[ 6.52414397,  4.33674962],
         [ 6.14284537,  3.09887053],
         [ 5.37677372,  7.50444614]],

        [[ 6.1225886 ,  3.3192871 ],
         [ 5.93006553,  1.95986379],
         [ 4.38583683,  7.388613  ]]],


       [[[ 4.68851114,  7.26166459],
         [ 3.57800094,  6.98710315],
         [ 7.20373333,  5.38633794]],

        [[ 6.91614031,  4.26511325],
         [ 6.74001646,  2.84167726],
```

```
 [ 4.7891852 ,  8.20297477]],

 [[ 6.69150931,  2.49794405],
  [ 6.89045511,  0.863456  ],
  [ 3.15897946,  8.00158859]]],


[[[ 6.40945058,  0.76002819],
  [ 6.5960878 , -0.0698415 ],
  [ 3.04148456,  3.76233945]],

 [[ 6.52137912,  5.24614314],
  [ 5.51461142,  4.31375332],
  [ 7.87237581,  6.82201766]],

 [[ 5.56851484,  6.11785216],
  [ 4.27055427,  5.28945591],
  [ 8.47204259,  6.92181322]]]])
```

Part 3. The actual original tensor was:

$$X_{1,1} = \begin{pmatrix} 4 & 0 & 9 \\ 7 & 9 & 9 \\ 4 & 8 & 5 \end{pmatrix} X_{2,1} = \begin{pmatrix} 7 & 8 & 2 \\ 1 & 5 & 8 \\ 7 & 9 & 2 \end{pmatrix} X_{3,1} = \begin{pmatrix} 7 & 9 & 4 \\ 10 & 1 & 2 \\ 1 & 5 & 8 \end{pmatrix} X_{1,2} = \begin{pmatrix} 6 & 5 & 1 \\ 3 & 3 & 5 \\ 1 & 8 & 7 \end{pmatrix} X_{2,2} = \begin{pmatrix} 8 & 2 & 3 \\ 4 & 3 & 3 \\ 2 & 4 & 6 \end{pmatrix}$$

$$X_{3,2} = \begin{pmatrix} 6 & 6 & 8 \\ 5 & 9 & 8 \\ 3 & 9 & 5 \end{pmatrix}$$

Calculate the MSE for both the CP and Tucker decompositions. Briefly discuss (2-3 sentences should be sufficient) the difference, especially regarding the relative reduction of features for each method.

[15]:
```
x11 = np.array([[4,0,9],
       [7,9,9],
       [4,8,5]])

x21 = np.array([[7,8,2],
       [1,5,8],
       [7,9,2]])

x31 = np.array([[7,9,4],
       [10,1,2],
       [1,5,8]])

x12 = np.array([[6,5,1],
       [3,3,5],
       [1,8,7]])
```

```
x22 = np.array([[8,2,3],
       [4,3,3],
       [2,4,6]])

x32 = np.array([[6,6,8],
       [5,9,8],
       [3,9,5]])

x = np.zeros((3,3,3,2))


x[:,:,0,0] = x11
x[:,:,1,0] = x21
x[:,:,2,0] = x31
x[:,:,0,1] = x12
x[:,:,1,1] = x22
x[:,:,2,1] = x32
```

0.1.7 CP MSE

[16]:
```
((x - X_tl)**2).mean()
```

[16]: 5.033737515123824

0.2 Tucker MSE

[17]:
```
((x - result_tl)**2).mean()
```

[17]: 4.927798012475143

1 Result

The original Tensor had 54 parameters. After CP Decomposition, the tensor was reduced to 26 parameters from the orignal 54. From these parameters, we achieved a reconstruction MSE of ~5. Conversely after Tucker decomposition, we were left with 38 parameters, a reduction of 16 parameters. The reconstruction MSE was ~4.9. With ~0.1 differene in MSE, CP would be a better choice due to the fewer amount of parameters

[ ]:
```

```

Q2

June 17, 2020

1 Question 2. Multilinear Algebra

Given

$$A * B = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, A * C = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}, C * D + D = \begin{pmatrix} 6 & 4 \\ 16 & 10 \end{pmatrix}, y = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}^T \tag{1}$$

find the vector of coefficients $\hat{\beta}$ by solving the following optimization problem:

$$\hat{\beta} = argmin_{\beta} \| y - \{ [(A \otimes C)^T * (B^T \otimes A^T)] \; [(B \odot C) * (A \odot D) + A * B \odot D] \} \beta \|_2^2 \tag{2}$$

Simplify the above expression to an appropriate form before solving the optimization problem.

Hint: $(A \otimes B) * (C \otimes D) = (A * C) \otimes (B * D); (A \odot B) * (C \odot D) = (A * C) \odot (B * D)$

[1]:
```
import numpy as np
from tensorly.tenalg import khatri_rao
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))
```

```
<IPython.core.display.HTML object>
```

2 *Note: Following section is for notes and references. The answer is below this section*

3 Hadamard Product

Element wise matrix multiplication E.g.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \qquad B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \tag{3}$$

$$A * B = \begin{pmatrix} 1 * 5 & 2 * 6 \\ 3 * 7 & 4 * 8 \end{pmatrix} \qquad = \begin{pmatrix} 5 & 12 \\ 21 & 32 \end{pmatrix} \tag{4}$$

Code

```
import numpy as np

A = np.array([[1,2],[3,4]])
B = np.array([[5,6],[7,8]])

ham = A*B
```

3.1 # Kronecker Product

Denoted $A \otimes B$ where A $\in \mathbb{R}^{IxJ}$ and B $\in \mathbb{R}^{KxL}$. The result is a matrix of size (IK) x (JL) and defined by:

$$\begin{pmatrix} a_{11}B & a_{12}B & ... & a_{1J}B \\ a_{21}B & a_{22}B & ... & a_{2J}B \\ ... & ... & ... & ... \\ a_{I1}B & a_{I2}B & ... & a_{IJ}B \end{pmatrix} \tag{5}$$

$$A \otimes B = \begin{pmatrix} 1*B & 2*B \\ 3*B & 4*B \end{pmatrix} = \begin{pmatrix} 5 & 6 & 10 & 12 \\ 7 & 8 & 14 & 16 \\ 15 & 18 & 20 & 24 \\ 21 & 24 & 28 & 32 \end{pmatrix} \tag{6}$$

Code

```
import numpy as np

A = np.array([[1,2],[3,4]])
B = np.array([[5,6],[7,8]])

kron = np.kron(A,B)
```

3.2 # Khati-Rao Product

"Matching columnwise" Kronecker product. Denoted $A \odot B$ where A $\in \mathbb{R}^{IxK}$ and B $\in \mathbb{R}^{JxK}$. A $\odot$ B is a matrix of size (IJ) x (K) and computed by

$$A \odot B = [a_1 \otimes b_1 \qquad a_2 \otimes b_2 \quad .... \quad a_k \otimes b_k] \tag{7}$$

$$A \odot B = \begin{pmatrix} 1*5 & 2*6 \\ 1*7 & 2*8 \\ 3*5 & 4*6 \\ 3*7 & 4*8 \end{pmatrix} \qquad = \begin{pmatrix} 5 & 12 \\ 7 & 16 \\ 15 & 24 \\ 21 & 32 \end{pmatrix} \tag{8}$$

Code

```
from tensorly.tenalg import khatri_rao

A = np.array([[1,2],[3,4]])
B = np.array([[5,6],[7,8]])

kr = khatri_rao([A,B])
```

3.3 # Answer:

A few things to note first

$$I. \quad **A*B** \in \mathbb{R}^{2x2}, \quad **A*C** \in \mathbb{R}^{2x2} \quad \therefore A \in \mathbb{R}^{2x2},\ B \in \mathbb{R}^{2x2}\ , and\ \ C \in \mathbb{R}^{2x2} \tag{9}$$

$$II. \quad (A \otimes B)^T = (A^T \otimes B^T) \tag{10}$$

$$III. \quad (A * C) = (C * A)\ -\ commutative\ property\ holds\ for\ Hadamard\ product \tag{11}$$

$$IV. \quad (A \odot B) + (A \odot C) = A \odot (B + C) - distributive\ property\ holds\ for\ Khatri\ Rao\ product \tag{12}$$

$$\hat{\beta} = argmin_{\beta} \|y - \{[(A \otimes C)^T * (B^T \otimes A^T)]\ [(B \odot C) * (A \odot D) + A * B \odot D]\}\beta\|_2^2 \qquad (1) \tag{13}$$

Rearraning the elements in the 1st square bracket of (1) according to item (II.) above and applying the hint $(A \otimes B) * (C \otimes D) = (A * C) \otimes (B * D)$

$$\hat{\beta} = argmin_{\beta} \|y - \{[(A^T \otimes C^T) * (B^T \otimes A^T)]\ [(B \odot C) * (A \odot D) + A * B \odot D]\}\beta\|_2^2 \qquad = argmin_{\beta} \|y - \{[(A^T * B^T) \otimes (C^T \tag{14}$$

Following the commutative property for Hadamard product as noted above in (III.) and reapplying item (II.) equation (2) can be written as

$$= argmin_{\beta} \|y - \{[(A * B)^T \otimes (A * C)^T]\ [(B \odot C) * (A \odot D) + A * B \odot D]\}\beta\|_2^2 \qquad (3) \tag{15}$$

Rearanging elements in the second square bracket of (3) according to the hint $(A \odot B) * (C \odot D) = (A * C) \odot (B * D)$

$$= argmin_{\beta} \|y - \{[(A * B)^T \otimes (A * C)^T]\ [(B * A) \odot (C * D) + A * B \odot D]\}\beta\|_2^2 \qquad (4) \tag{16}$$

Notating $(A * B) = (B * A) = \gamma$ (4) becomes

$$= argmin_{\beta} \|y - \{[(A * B)^T \otimes (A * C)^T] \, [\gamma \odot (C * D) + \gamma \odot D]\}\beta\|_2^2 \qquad (5) \qquad (17)$$

Notating item (IV.) above and factoring out γ

$$= argmin_{\beta} \|y - \{[(A * B)^T \otimes (A * C)^T] \, [\gamma \odot ((C * D) + D) \,]\}\beta\|_2^2 \qquad (6) \qquad (18)$$

Replacing γ with $\gamma = (A * B)$ in (6) we arrive at

$$\hat{\beta} = argmin_{\beta} \|y - \{[(A * B)^T \otimes (A * C)^T] \, [(A * B) \odot ((C * D) + D) \,]\}\beta\|_2^2 \qquad (7) \qquad (19)$$

$$\boxed{\hat{\beta} = argmin_{\beta} \|y - X\beta\|_2^2 \qquad \therefore X = \{[(A * B)^T \otimes (A * C)^T] \, [(A * B) \odot ((C * D) + D) \,]\}} \qquad (8) \qquad (20)$$

[2]:
```
ab = np.array([[1,2],[3,4]])
ac = np.array([[5,6],[7,8]])
cdd = np.array([[6,4],[16,10]])
y = np.array([1,2,3,4])
```

3.3.1 Using eqn. (8) above, create X matrix

[3]:
```
#left square bracket - Kronecker product
M1 = np.kron(ab.T, ac.T)
```

[4]:
```
#right square bracket - khatri rao prodcut
M2 = khatri_rao([ab, cdd])
```

[5]:
```
# X - matrix
X = M1.dot(M2)
```

3.3.2 Using numpy to solve OLS for beta

[6]:
```
betas_np = np.linalg.lstsq(X,y, rcond=None)[0]
betas_np
```

[6]:
```
array([-0.0309884 ,  0.03603101])
```

3.3.3 Manual calculation of OLS for sanity check $\hat{\beta} = (X^T X)^{-1} X^T y$

[7]:
```
betas_ols = np.linalg.pinv(X.T.dot(X)).dot(X.T.dot(y))
betas_ols
```

[7]:
```
array([-0.0309884 ,  0.03603101])
```

$$\boxed{\hat{\beta} = \begin{pmatrix} -0.0309884 \\ 0.03603101 \end{pmatrix}} \tag{21}$$

[ ]:
```

```

Q3

June 16, 2020

1 Question 3. Image Classification

Dimensionality reduction, feature extraction and selection are crucial parts of high multidimensional data analysis. Consider a set of K training samples $X^{(k)} \in R^{I_1 x I_2 x \dots x I_N}$, (k=1,2, … , K) corresponding to C categories/classes, and a set of test data $X^t \in R^{I_1 x I_2 x \dots x I_N}$, (t=1,2, … , T). The challenge is to find appropriate labels for the test data. The classification algorithm can be generally performed in the following steps:

1. Find a set of basis matrices and the corresponding features from the training data $X^{(k)}$. The relation of a sample $X^{(k)}$ and basis factors can be expressed as:

$$X^{(k)} \approx G^{(k)} x_1 A^{(1)} x_2 A^{(2)} \dots x_N A^{(N)} (k = 1, 2, \dots K) \tag{1}$$

Where the core tensor $G^{(k)} \in R^{J_1 x J_2 x \dots x J_N}$ representing features of a much lower dimension than the training data $X^{(k)}$. In other words, the core tensor $G^{(k)}$ consists of features of $X^{(k)}$ in subspace $A^{(n)}$.

2. Perform feature extraction for the test samples $X^{(t)}$ using the basis factors found for the training data (using a projected filter).

$$X^{(t)} x_1 A^{(1)T} \dots x_N A^{(N)T} \tag{2}$$

3. Perform classification by comparing the test features with the training features.

You are given 28 training images, train1.jpg through train28.jpg. The first 14 images correspond to cats, and the remaining images correspond to birds. There are two classes: cats and birds. The labels for the images can be found in the file train_lab.mat. Your job will be to classify 12 new images, Test1.jpg through Test12.jpg. Use the training features to train a random forest with 100 trees. Note that you will need to vectorize the training features.

[1]:
```
from scipy.io import loadmat
import matplotlib.pyplot as plt
import numpy as np
import tensorly as tl
from tensorly.tenalg import khatri_rao
from tensorly.decomposition import tucker
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import os
```

```
import re
import cv2
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))
```

<IPython.core.display.HTML object>

[2]:
```
# Helper functions for properly sorting the train and test images by name
def atoi(text):
    return int(text) if text.isdigit() else text

def natural_keys(text):
    '''
    alist.sort(key=natural_keys) sorts in human order
    http://nedbatchelder.com/blog/200712/human_sorting.html
    (See Toothy's implementation in the comments)
    '''
    return [ atoi(c) for c in re.split(r'(\d+)', text) ]
```

[3]:
```
#list of image paths for train and test images
train = [os.path.join('train', x) for x in sorted(os.listdir('train'),␣
 ↪key=natural_keys)]
test = [os.path.join('test', x) for x in sorted(os.listdir('test'),␣
 ↪key=natural_keys)]
```

[4]:
```
#load train and test labels
train_labs = loadmat('train_lab.mat')['train']
test_labs = loadmat('test_lab.mat')['test']
```

1.0.1 Part 1. Read and convert all images into gray scale. Form a third-order tensor using the training data and apply Tucker decomposition with $R_1 = 10; R_2 = 10; R_3 = 28$. Predict the labels for the images on the test set. Report the classification error.

[5]:
```
#load and convert to grayscale
grays_train = [cv2.cvtColor(plt.imread(x), cv2.COLOR_RGB2GRAY) for x in train]
grays_test = [cv2.cvtColor(plt.imread(x), cv2.COLOR_RGB2GRAY) for x in test]
```

[6]:
```
#Form a third order tensor of the gray images
X_train = np.dstack(grays_train)
X_test = np.dstack(grays_test)
```

[7]:
```
#apply Tucker Decmoposition
core, factors = tucker(tl.tensor(X_train, dtype=tl.float32), ranks=[10,10,28])
```

[8]:
```
#abosrb core into the 3rd dimension factor since only the first 2 factors are␣
↪needed
core=tl.tenalg.mode_dot(core, factors[-1], mode=2)
#vectorize the core tensor
core = core.reshape(-1,28).T
```

[9]:
```
#create and train the RF classifier
clf = RandomForestClassifier()
clf.fit(core, train_labs.ravel())
```

[9]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

[10]:
```
#transform test tensor
ttc = tl.tenalg.mode_dot(tl.tenalg.mode_dot(X_test, factors[0].T, mode=0),␣
↪factors[1].T, mode=1)
#vectorize the tensor
ttc=ttc.reshape(-1,12).T
```

[11]:
```
#predict on test set
preds = clf.predict(ttc)
```

[12]:
```
#calculate accuracy on test set
acc = accuracy_score(test_labs.ravel(), preds)
```

[13]:
```
print('Accuracy on test set: {}'.format(acc))
print('{} out of {} correctly predicted'.format(len(preds) - sum(test_labs.
↪ravel() - preds), len(preds)))
```

```
Accuracy on test set: 0.9166666666666666
11 out of 12 correctly predicted
```

1.0.2 Part 2. Read all images in RGB format. Form a fourth-order tensor using the training data and apply Tucker decomposition with $R_1 = 10; R_2 = 10; R_3 = 3; R_4 = 28$. Predict the labels for the images on the test set. Report the classification error.

[14]:
```
#Create 4th order tensor with the RGB images
rgb_train = [plt.imread(x) for x in train]
rgb_test = [plt.imread(x) for x in test]
```

[15]:
```
X_train = np.stack(rgb_train, axis=3)
X_test = np.stack(rgb_test, axis=3)
```

[16]:
```
#apply Tucker Decmoposition
core, factors = tucker(tl.tensor(X_train, dtype=tl.float32), ranks=[10,10,3,28])
```

[17]:
```
#abosrb core into the 4th dimension factor since only the first 3 factors are␣
↪needed
core=tl.tenalg.mode_dot(core, factors[-1], mode=3)
#vectorize the core tensor
core = core.reshape(-1, 28).T
```

[18]:
```
#create and train the RF classifier
clf = RandomForestClassifier()
clf.fit(core, train_labs.ravel())
```

[18]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

[19]:
```
#transform test tensor
ttc = tl.tenalg.mode_dot(tl.tenalg.mode_dot(tl.tenalg.mode_dot(X_test,␣
↪factors[0].T, mode=0), factors[1].T, mode=1), factors[2].T, mode=2)
#vectorize the tensor
ttc=ttc.reshape(-1,12).T
```

[20]:
```
#predict on test set
preds = clf.predict(ttc)
```

[21]:
```
#calculate accuracy on test set
acc = accuracy_score(test_labs.ravel(), preds)
```

[22]:
```
print('Accuracy on test set: {}'.format(acc))
print('{} out of {} correctly predicted'.format(len(preds) - sum(test_labs.
 ↪ravel() - preds), len(preds)))
```

```
Accuracy on test set: 0.9166666666666666
11 out of 12 correctly predicted
```

Q4

June 16, 2020

1 Question 4. Heat transfer process

Consider a heat transfer process that follows the following equation:

$$\frac{\partial S(x,y,t)}{\partial t} = \alpha(\frac{\partial^2 S}{\partial x^2} + \frac{\partial^2 S}{\partial y^2}) \tag{1}$$

where $0 \leq x, y \leq 0.05$ represents the location of each pixel, α is the thermal diffusivity coefficient, and t is the time frame. The initial boundary conditions are set such that $S|_{t=1} = 0$ and $S|_{x=0} = S|_{x=0.05} = S|_{y=0} = S|_{y=0.05} = 1$. At each time , the image is recorded at locations $x = \frac{j}{n+1}, y = \frac{k}{n+1}, j, k = 1, ..., n$, resulting in an n x n matrix. Here we set n=21 and t=1, … , 10, which leads to 10 images of size 21 x 21, that can be represented as a 21x21x10 tensor.

The thermal diffusivity coefficient depends on the material being heated. In the dataset heatT.mat, we have tensor 1, tensor 2 and tensor 3 corresponding to a heat transfer process in material 1, material 2 and material 3, respectively.

[1]:
```
from scipy.io import loadmat
import matplotlib.pyplot as plt
import numpy as np
np.set_printoptions(edgeitems=30, linewidth=100000)
import tensorly as tl
from tensorly import unfold
from tensorly.tenalg import inner
from tensorly.decomposition import parafac
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import os
import re
import cv2
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))
```

```
<IPython.core.display.HTML object>
```

1.0.1 Part 1. Try different ranks for CP decomposition and use AIC to choose the optimal one.

[2]:
```
#load data from .mat file
data = loadmat('heatT.mat')
```

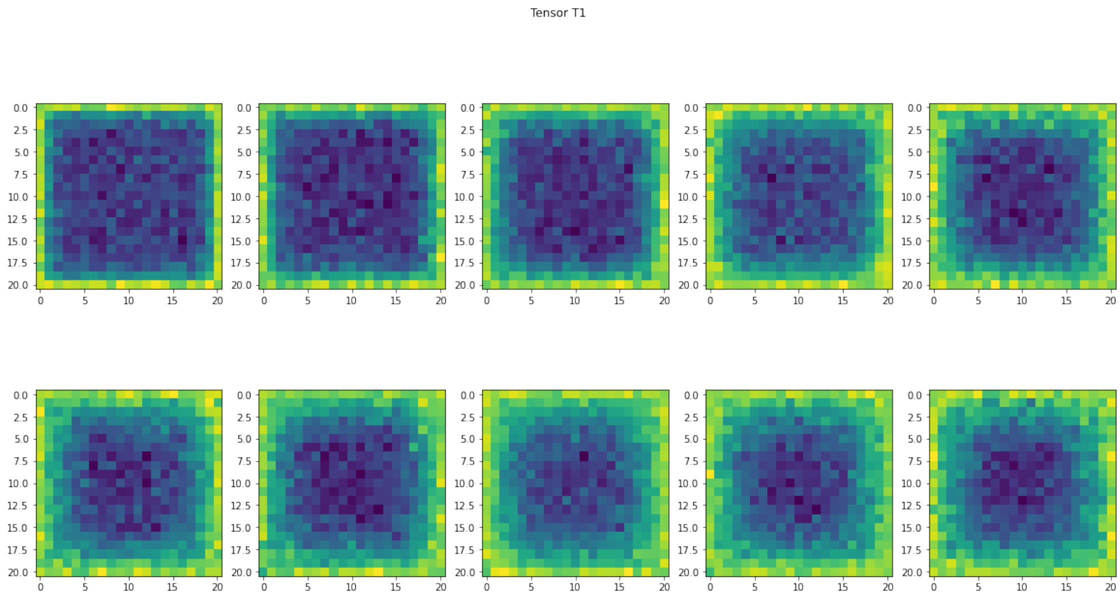
[3]:
```
#extract data from nested arrays from loading of the .mat file
T1 = data['T1'][0][0][0]
T2 = data['T2'][0][0][0]
T3 = data['T3'][0][0][0]

print('T1 shape', T1.shape, '\nT2 shape', T2.shape, '\nT3 shape', T3.shape)
```

```
T1 shape (21, 21, 10)
T2 shape (21, 21, 10)
T3 shape (21, 21, 10)
```

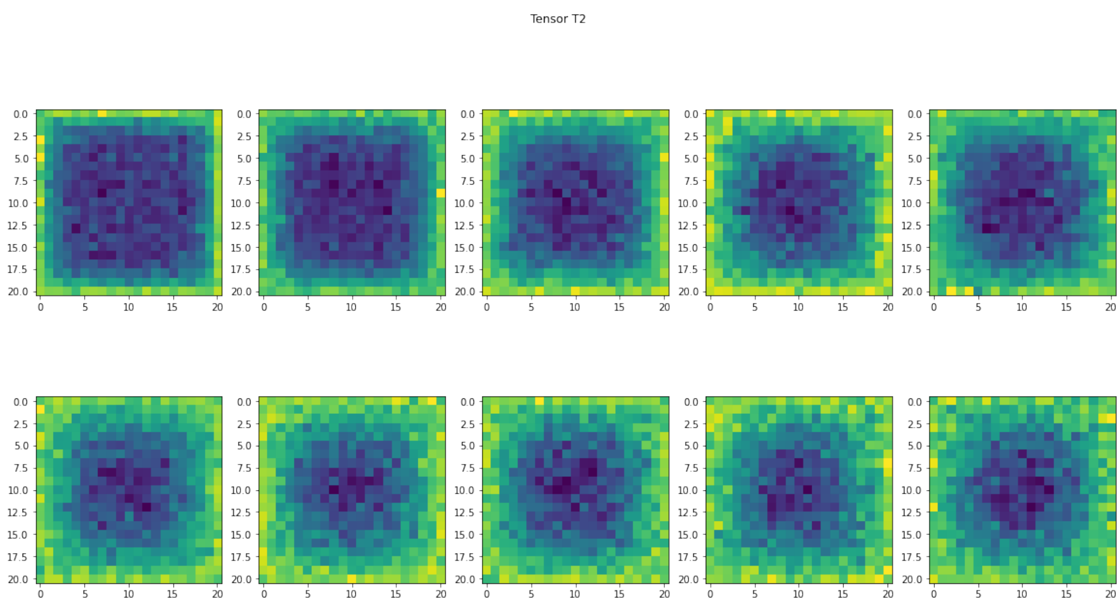
[4]:
```
#function to help view the data
def multi_plot(T, title):
    plt.subplots(nrows=2, ncols=5, figsize=(20,10))

    for i in range(1, T.shape[2]+1):
        plt.subplot(2,5,i)
        plt.imshow(T[:,:,i-1])

    plt.suptitle(title)

    plt.show()
```

1.0.2 Visualize each component of the Tensors

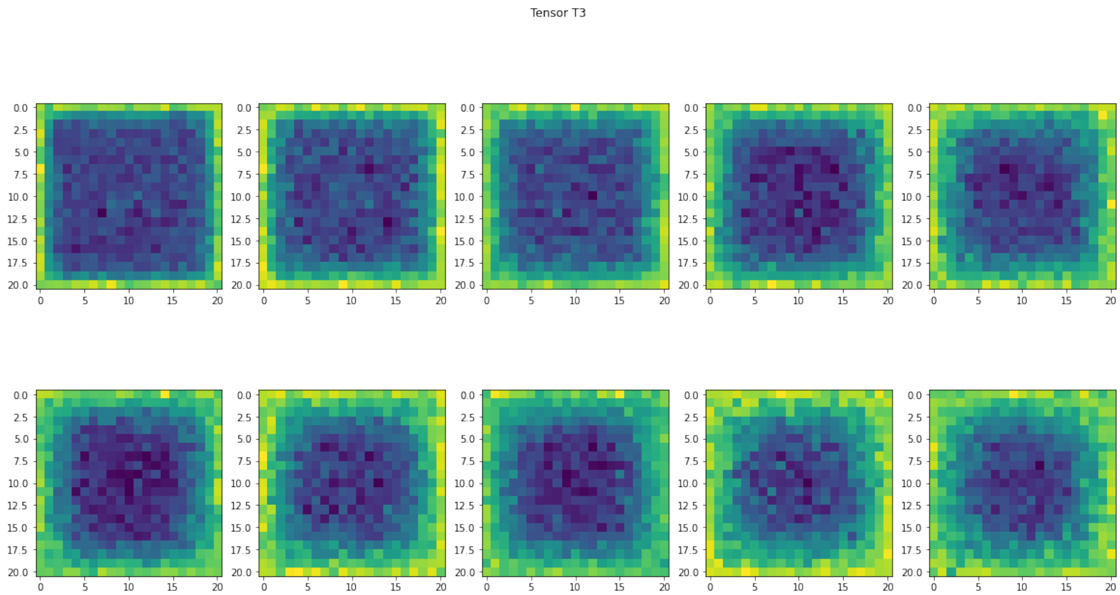
[5]:
```
multi_plot(T1, 'Tensor T1')
```



[6]:
```python
multi_plot(T2, 'Tensor T2')
```



[7]:
```python
multi_plot(T3, 'Tensor T3')
```



[8]:
```
#convert numpy array to tensorly tensors
T1 = tl.tensor(T1, dtype=tl.float32)
T2 = tl.tensor(T2, dtype=tl.float32)
T3 = tl.tensor(T3, dtype=tl.float32)
```

[9]:
```
#function to calculate all AICs for each tensor
def find_lowest(t1, t2, t3):
    #initialize lists to store errors for each rank of decomposition
    t1_aic = []
    t2_aic = []
    t3_aic = []

    #try out up to 20 different ranks
    for r in range(1,21):
        #weights, and factors from each decomp on each tensor
        w1, f1 = parafac(t1, r, normalize_factors=True)
        w2, f2 = parafac(t2, r, normalize_factors=True)
        w3, f3 = parafac(t3, r, normalize_factors=True)

        #reconstruct from factors
        x1 = tl.kruskal_to_tensor((w1,f1))
        x2 = tl.kruskal_to_tensor((w2,f2))
        x3 = tl.kruskal_to_tensor((w3,f3))

        #calculate error in reconstruction
        err1 = err(t1, x1)
        err2 = err(t2, x2)
```

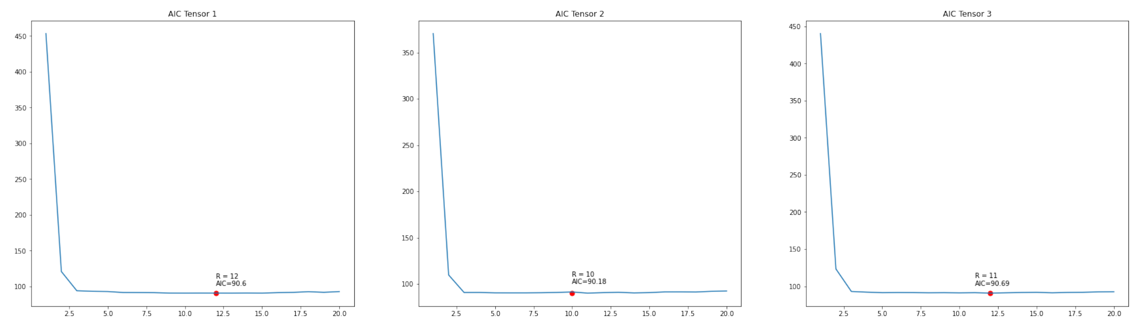
```
        err3 = err(t3, x3)

        #calculate AIC using rank as the penalization term instead of number of␣
↪parameters as suggested by prof on Piazza
        t1_aic.append(AIC(err1, r))
        t2_aic.append(AIC(err2, r))
        t3_aic.append(AIC(err3, r))

    #return the 3 arrays each containing 20 different AIC values
    return t1_aic, t2_aic, t3_aic
```

[10]:
```
def err(orig, calc):
    diff = orig - calc
    err = (diff**2).sum()
    return err
```

[11]:
```
def AIC(err, r):
    return 2*err + (2*r)
```

[12]:
```
#get AIC values
a1,a2,a3=find_lowest(T1,T2,T3)
```

[13]:
```
plt.subplots(nrows=1, ncols=3, figsize=(30,8))
plt.subplot(131)
plt.plot(range(1,len(a1)+1),a1)
plt.scatter(np.argmin(a1), min(a1), color='r', s=50)
plt.text(np.argmin(a1), min(a1)+10, 'R = {}\nAIC={}'.format(np.argmin(a1),␣
↪round(min(a1),2)))
plt.title('AIC Tensor 1')

plt.subplot(132)
plt.plot(range(1,len(a2)+1),a2)
plt.scatter(np.argmin(a2), min(a2), color='r', s=50)
plt.text(np.argmin(a2), min(a2)+10, 'R = {}\nAIC={}'.format(np.argmin(a2),␣
↪round(min(a2),2)))
plt.title('AIC Tensor 2')

plt.subplot(133)
plt.plot(range(1,len(a3)+1),a3)
plt.scatter(np.argmin(a1), min(a3), color='r', s=50)
plt.text(np.argmin(a3), min(a3)+10, 'R = {}\nAIC={}'.format(np.argmin(a3),␣
↪round(min(a3),2)))
plt.title('AIC Tensor 3')
plt.show()
```



1.0.3 Part 2. Use CP decomposition to decouple temporal and spatial patterns of the three materials in heat transfer processes. Plot the first 4 spatial and temporal patterns of tensor 1.

[14]:
```python
#use the optimal rank from Part 1
r1 = np.argmin(a1)
r2 = np.argmin(a2)
r3 = np.argmin(a3)
```

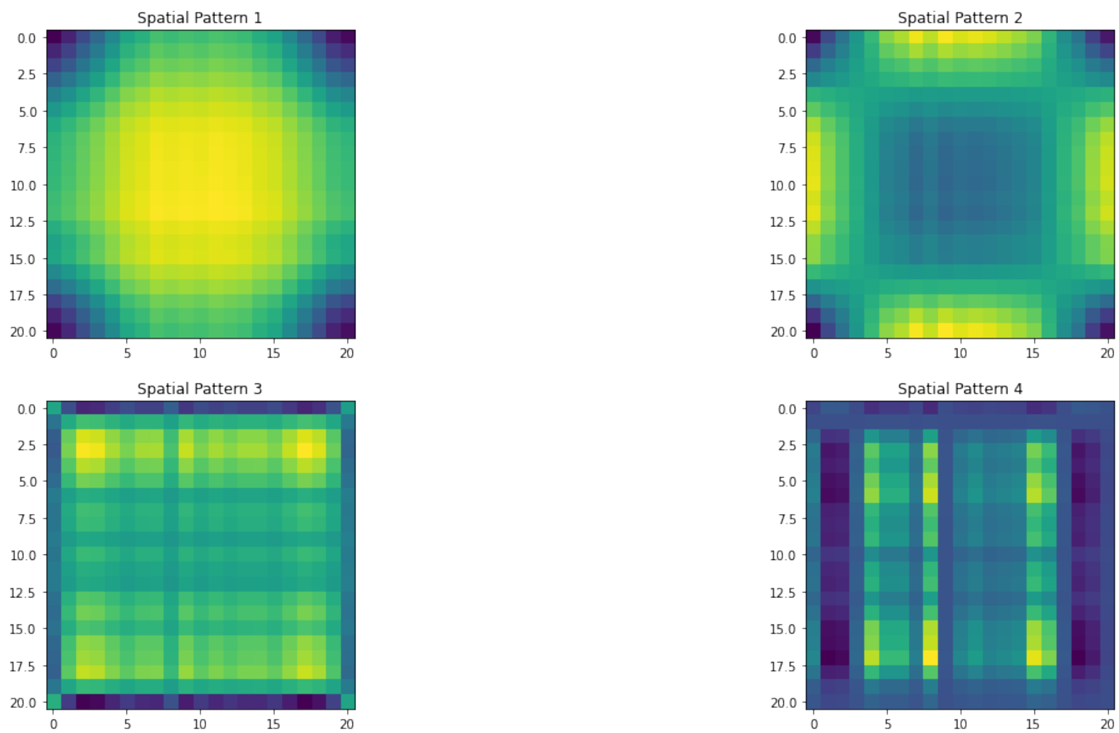
[15]:
```python
#use CP decomp using optimal ranks
w1, f1 = parafac(T1, r1, normalize_factors=True)
w2, f2 = parafac(T2, r2, normalize_factors=True)
w3, f3 = parafac(T3, r3, normalize_factors=True)
```

[16]:
```python
#weights and factor columns are not sorted in descending order - sort them in descending order
sorted_lambdas_and_indices = sorted(list(zip(range(len(w1)), w1)), key=lambda x: x[1], reverse=True)
sorted_idxs = [x[0] for x in sorted_lambdas_and_indices]
sorted_lambdas = [x[1] for x in sorted_lambdas_and_indices]
sorted_lambdas_and_indices
```

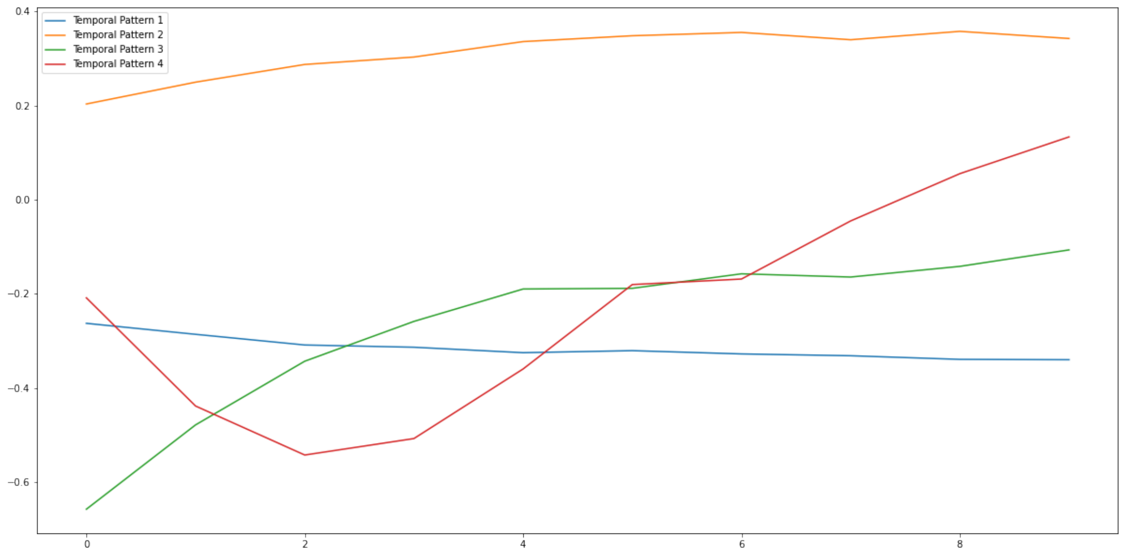
[16]:
```
[(0, 40.57829),
 (10, 9.616071),
 (1, 7.672068),
 (2, 3.868541),
 (11, 3.7067842),
 (7, 2.519711),
 (3, 1.8275048),
 (4, 1.452806),
 (8, 1.4356604),
 (5, 1.4246788),
 (9, 1.4000778),
 (6, 1.0527846)]
```

[17]:
```
#sort factor columns based of of weight orders
factor1 = f1[0][:,sorted_idxs]
factor2 = f1[1][:,sorted_idxs]
factor3 = f1[2][:,sorted_idxs]
```

[18]:
```
plt.subplots(nrows=2, ncols=2, figsize=(20,10))
for i in range(4):
    plt.subplot(2,2,i+1)
    plt.imshow((np.kron(factor1[:,i] , factor2[:,i]) * sorted_lambdas[i]).
 ↪reshape((21,21)))
    plt.title('Spatial Pattern {}'.format(i+1))

plt.show()
```



[19]:
```
plt.figure(figsize=(20,10))
for i in range(4):
    plt.plot(factor3[:,i], label='Temporal Pattern {}'.format(i+1))

plt.legend()
plt.show()
```


Temporal Pattern 1
Temporal Pattern 2
Temporal Pattern 3
Temporal Pattern 4
0.4
0.2
0.0
−0.2
−0.4
−0.6
0
2
4
6
8