Q4

July 17, 2020

[13]:
```
import numpy as np
import matplotlib.pyplot as plt
import cv2
from scipy.interpolate import BSpline
from scipy.linalg import sqrtm
from scipy.optimize import fminbound
from skimage.filters import threshold_otsu
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))
```

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

In this problem, we're going to use Sparse Smooth Decomposition to extract features rather than detect anomalies. Provided are two images of heatmaps of a GPU lid. One is at idle, and the other one is under load (training a CNN in to classify tensors of birds and cats.) We want to programmatically detect where heat spreads on the lid under load so engineers can design appropriate heatsinks and place thermal sensors on the die. Unfortunately, the temperature sensor we have is very noisy when the GPU is idle due to the temperature differentials being quite small. Therefore, we can't solely rely on image processing techniques from Module 2, such as simply subtracting the at-idle image from the at-load image and doing edge detection.

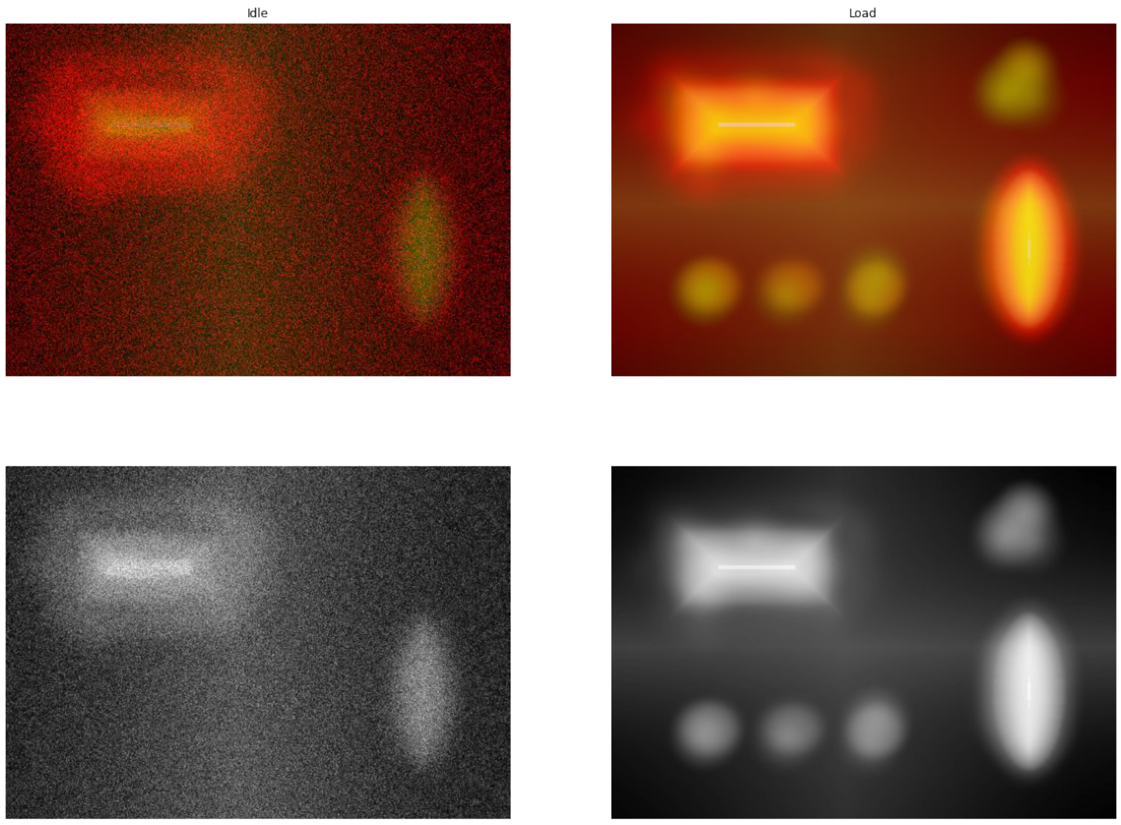
0.1 A: 10%)

Read in both images and convert to grayscale. To demonstrate why this would be an ugly problem with simple techniques, show a simple subtraction of the idle image from the load image as the deliverable for this part.

[2]:
```
idle = plt.imread('heat_idle.jpg')
load = plt.imread('heat_load.jpg')

idle_gray = cv2.cvtColor(idle, cv2.COLOR_RGB2GRAY)
load_gray = cv2.cvtColor(load, cv2.COLOR_RGB2GRAY)
```

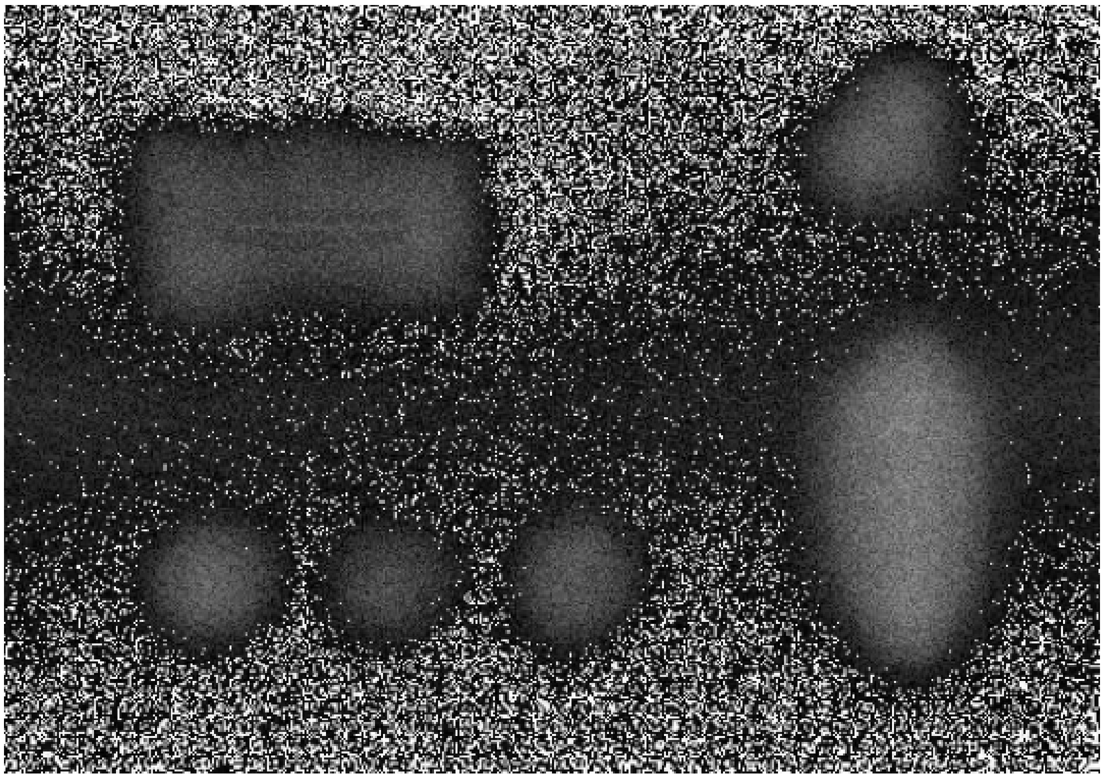
[3]:
```
plt.subplots(2,2,figsize=(20,15))
plt.subplot(221)
plt.imshow(idle)
plt.axis('off')
```

```python
plt.title('Idle')
plt.subplot(222)
plt.imshow(load)
plt.axis('off')
plt.title('Load')
plt.subplot(223)
plt.imshow(idle_gray, cmap='gray')
plt.axis('off')
plt.subplot(224)
plt.imshow(load_gray, cmap='gray')
plt.axis('off')
plt.show()
```



```python
[4]: subtracted = load_gray - idle_gray
```

```python
[5]: plt.figure(figsize=(10,10))
     plt.imshow(subtracted, cmap='gray')
     plt.axis('off')
     plt.show()
```



0.2 B: 40%)

Implement 2-D SSD. For purposes of this part, use the default parameters, as in the example code. (Eg., delta = 0.2, x and y knots = 6, anomaly knots = length/4.) As deliverables, include in your report the same output as from the example code. That is, the:

- combined image used
- decomposed mean
- decomposed features – we are interested in heat generated under load, so set values <0 to 0!

[6]:
```
'''
This code is refactored matlab code ported to Python done by
Johnathan Tay. refactored code can be found here:
https://github.gatech.edu/jtay6/IYSE8803-Examples-Py/blob/master/Module%207/
↪Examples7.py
'''

def BSplineBasis(x: np.array, knots: np.array, degree: int) -> np.array:
    '''Return B-Spline basis. Python equivalent to bs in R or the spmak/spval␣
↪combination in MATLAB.
    This function acts like the R command bs(x,knots=knots,degree=degree,␣
↪intercept=False)
```

```
    Arguments:
        x: Points to evaluate spline on, sorted increasing
        knots: Spline knots, sorted increasing
        degree: Spline degree.
    Returns:
        B: Array of shape (x.shape[0], len(knots)+degree+1).
    Note that a spline has len(knots)+degree coefficients. However, because the␣
↪intercept is missing
    you will need to remove the last 2 columns. It's being kept this way to␣
↪retain compatibility with
    both the matlab spmak way and how R's bs works.
    If K = length(knots) (includes boundary knots)
    Mapping this to R's bs: (Props to Nate Bartlett )
    bs(x,knots,degree,intercept=T)[,2:K+degree] is same as␣
↪BSplineBasis(x,knots,degree)[:,:-2]
    BF = bs(x,knots,degree,intercept=F) drops the first column so BF[,1:
↪K+degree] == BSplineBasis(x,knots,degree)[:,:-2]
    '''
    nKnots = knots.shape[0]
    lo = min(x[0], knots[0])
    hi = max(x[-1], knots[-1])
    augmented_knots = np.append(
        np.append([lo]*degree, knots), [hi]*degree)
    DOF = nKnots + degree + 1  # DOF = K+M, M = degree+1
    spline = BSpline(augmented_knots, np.eye(DOF),
                     degree, extrapolate=False)
    B = spline(x)
    return B


def thresh(x, t, tau):
    assert t in ['s', 'h']

    if t is 't':
        tmp = x.copy()
        tmp[np.abs(tmp) < tau] = 0
        return tmp
    else:
        return np.sign(x)*np.maximum(np.abs(x)-tau, 0)



def bsplineSmoothDecompauto(y, B, Ba, lam, gamma, maxIter=20, errtol=1e-6):
    def plus0(x): return np.maximum(x, 0)
    def norm(x): return np.linalg.norm(x, 2)
    sizey = y.shape
    ndim = len(y.squeeze().shape)
```

```
if ndim == 1:
    Lbs = 2*norm(Ba[0])**2
    X = np.zeros(Ba[0].shape[1])
    a = 1
    BetaA = X.copy()
elif ndim == 2:
    Lbs = 2*norm(Ba[0])**2*norm(Ba[1])**2
    X = np.zeros((Ba[0].shape[1], Ba[1].shape[1]))
    BetaA = X.copy()

if len(lam) == 1:
    lam = np.ones(ndim)*lam

SChange = 1e10
H = []
a = np.zeros_like(y)
C = []
Z = []

for idim in range(ndim):
    Li = sqrtm(B[idim].T@B[idim])
    Li = Li + 1e-8*np.eye(*Li.shape)
    Di = np.diff(np.eye(B[idim].shape[1]), 1, axis=0)
    tmp = np.linalg.pinv(Li.T)@(Di.T@Di)@np.linalg.pinv(Li)
    Ui, ctmp, _ = np.linalg.svd(tmp)
    C.append(np.diag(ctmp))
    Z.append(B[idim]@np.linalg.pinv(Li.T)@Ui)

iIter = 0
t = 1

while SChange > errtol and iIter < maxIter:
    iIter += 1
    Sold = a
    BetaSold = BetaA
    told = t
    def gcv(x): return splinegcv(x, y, C, Z, 0, [])

    if len(lam) == 0 and iIter == 1:
        lam = fminbound(gcv, 1e-2, 1e3)
        lam = lam*np.ones(ndim)

    # % %
    H = []
    for idim in range(ndim):
        L1 = C[idim].shape[0]
```

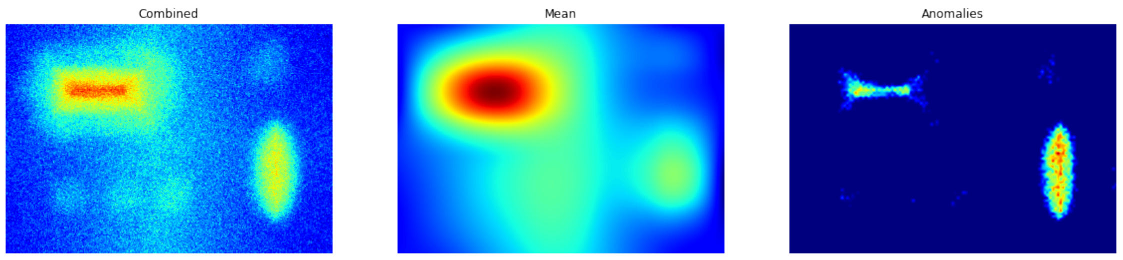
```
            o = np.ones(L1)+lam[idim]*np.diag(C[idim])
            tmp = Z[idim]@np.diag(1/o)@Z[idim].T
            H.append(tmp)
        if ndim == 1:
            yhat = H[0]@(y-a)
            BetaSe = X + 2/Lbs*Ba[0].T@(y - Ba[0]@X - yhat)
        elif ndim == 2:
            yhat = H[0]@(y-a)@H[1]
            BetaSe = X + 2/Lbs*Ba[0].T@(y - Ba[0]@X@Ba[1].T - yhat)@Ba[1]

        maxYe = np.abs(BetaSe).max()

        # %
        if not gamma and iIter % 3 == 1:
            gamma = threshold_otsu(np.abs(BetaSe)/maxYe)*maxYe*Lbs

        # change 'h' to 's' for softthresholding
        BetaA = thresh(BetaSe, 'h', gamma/Lbs)
        if ndim == 1:
            a = Ba[0] @BetaA
        elif ndim == 2:
            a = Ba[0] @BetaA@ Ba[1].T
        t = (1+(1+4*told**2)**0.5)/2

        if iIter == 1:
            X = BetaA
        else:
            X = BetaA+(told-1)/t*(BetaA-BetaSold)

        SChange = a-Sold
        SChange = (SChange**2).sum()

    return yhat, a


def splinegcv(lam, Y, C, Z, nmiss, W):
    # % Estimate Generalized Cross-validation value

    ndim = len(np.squeeze(Y).shape)
    H = []
    dfi = np.zeros(ndim)
    for idim in range(ndim):
        # print(ndim,idim)
        L1 = C[idim].shape[0]
        # o = np.ones(L1)+lam*np.diag(C[idim])
        o = 1+lam*np.diag(C[idim])
        tmp = Z[idim]@np.diag(1/o)@Z[idim].T
        H.append(tmp)

        dfi[idim] = sum(1/(1+lam*np.diag(C[idim])))

    df = np.product(dfi)
    if ndim == 1:
        Yhat = H[0]@Y
    elif ndim == 2:
        # print(H[0].shape,H[1].shape,Y.shape)
        Yhat = H[0]@Y@H[1]
    elif ndim >= 3:
        raise NotImplementedError
        # Yhat = double(ttm(tensor(Y),H));

    if not W:
        RSS = ((Y-Yhat)**2).sum()
    else:
        diff = Y-Yhat
        RSS = (diff*W*diff).sum()

    n = len(Y)
    GCVscore = RSS/(n-nmiss)/(1-df/n)**2
    return GCVscore
```

[7]:
```
def ssd(delta=0.2, kx=6, ky=6, snk=4, lam=[], gam=[], maxIter=20, errtol=1e-6):

    nx, ny = idle_gray.shape

    Y0 = idle_gray
    A0 = load_gray

    Y = Y0 + delta*A0

    B1 = BSplineBasis(np.arange(nx), np.linspace(0, nx-1, kx), 2)[:,:-2]
    B2 = BSplineBasis(np.arange(ny), np.linspace(0, ny-1, ky), 2)[:,:-2]

    skx = int(np.round(nx/snk))
    sky = int(np.round(ny/snk))

    Bs1 = BSplineBasis(np.arange(nx), np.linspace(0, nx-1, skx), 1)[:, :-2]
    Bs2 = BSplineBasis(np.arange(ny), np.linspace(0, ny-1, sky), 1)[:, :-2]

    y = Y.copy()
    B = (B1, B2)
    Ba = (Bs1, Bs2)

    yhat,a = bsplineSmoothDecompauto(Y, B, Ba, [], [])
    yhat[yhat<0]=0
    a[a<0]=0

    return yhat, a, Y
```

[8]:
```
def plotit(cmap='jet'):
    plt.subplots(1,3,figsize=(20,15))
    plt.subplot(131)
    plt.imshow(Y, cmap=cmap)
    plt.title('Combined')
    plt.axis('off')
    plt.subplot(132)
    plt.imshow(yhat, cmap=cmap)
    plt.title("Mean")
    plt.axis('off')
    plt.subplot(133)
    plt.imshow(a, cmap=cmap)
    plt.axis('off')
    plt.title('Anomalies')
    plt.show()
```

[9]:
```
yhat, a, Y = ssd()
```

[10]:
```
plotit('jet')
```



0.3 C: 50%)

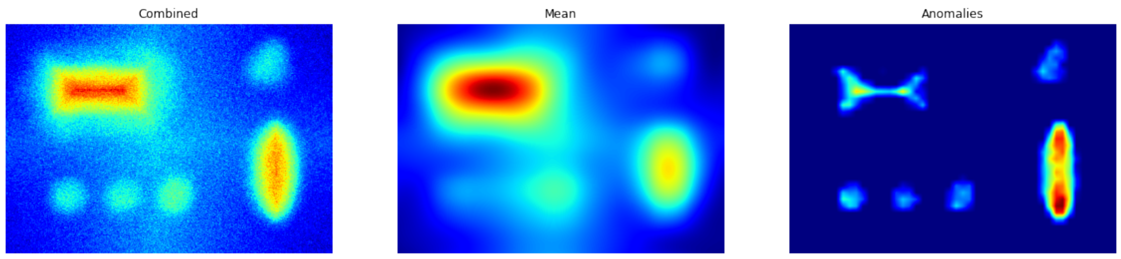
The default parameters do quite well, but we can do better! Play with all available parameters to generate the best separation you can. Remember, your goal is to capture the load heat as sparse "anomaly" in the decomposition. In other words, you (probably) don't want the spots that are only hot under load to show up in the mean. For this part, include in your report your:

- combined image used (the "delta" used when combining the images is fair game)
- decomposed mean
- decomposed features – again, set values <0 to 0!

- a brief discussion of your methodology; say what you tried, wat did (or didn't) work, and why you chose what you finally chose.

```
[11]: yhat, a, Y = ssd(delta=0.7, kx=10, ky=10, snk=10, lam=[], gam=[0.1],␣
       ↪maxIter=10000, errtol=1e-15)
```

```
[12]: plotit('jet')
```



0.3.1 Results

In attempt to find the best parameters, the main item focused on was "..you don't want the spots that are only hot under load to show up in the mean." Because of this, I focused on a combined image the sufficiently isolated all the hot spots. The parameter that affected this the most was delta. A higher delta allowed for more "load" representation to fill in the combined image and distinctly isolate hot regions. A value of 0.7 gave the best result without being too overearing in the "loaded" regime. Next I focused on a mean image that sufficiently isolated the hot spots as well. I found that altering the number of knots both in the x and y direction affected how "connected" the hot regions were. A value of 10 for both knows was sufficent for this. Finally I worked on refining the resolution of the Anomalies by adjusting the snk parameter.

- delta = 0.7
- kx = ky = 10
- snk = 10