Biometría practical work

Joakim Olsen

May 2019

Introduction

For the practical assignment, I have implemented eigenfaces and fisherfaces for face recognition. I have used the ORL database (400 images of 40 individuas), as well as a part of the Yale database (100 images with different light setting from 10 individuals) to test the performance of the algorithms. Finally, I have implemented local light normalization, to see how this can improve the performance on images with poor lighting. I have not implemented local histogram equalization, since the eigenface and fisherface exercises together give 9 points, and I therefore figured doing just the local normalization will give the suffucient amount of points (4/2 = 2). Also, the local normalization is sufficient to analyse the improvement of the performance, and thus also implementing local histogram equalization didn't seem to add that much more insight for the benefit of my own learning.

1 Eigenfaces

Eigenfaces is implemented the following way: The data is split into a training set X_{tr} and a test set X_{te}, both with n samples, thus they are both $n \times d$ - matrices, which seems to be transposed compared to in the class slides, but which I prefer when implementing in python. Then μ_{tr} is calculated as the average of X_{tr}, a d - dimensional vector. We then have $A^t = X_{tr} - \mu_{tr}$, denoted as the transpose of A since A^t becomes a $n \times d$ matrix, thus, A is a $d \times n$ matrix, as in the class slides. From here there are two ways to continue:

1. We calculate $C = \frac{1}{n}AA^t$, a $d \times d$ matrix. Then the eigenvectors B of this is calculated so that $CB = B\Delta$. For the images, the number of dimensions d is very high. It will therefore be very computationally heavy to calculate B. We should therefore consider to find a smarter way to find B.

2. We calculate $C' = \frac{1}{d}A^tA$, a $n \times n$ matrix. Then we calculate the eigenvectors of C', B', so that $C'B' = B'\Delta'$. As shown in the class slides, we can now obtain B and Δ as $B = AB'$ and $\Delta = \frac{d}{n}\Delta'$. Note that B is now a $d \times n$ - matrix, so there is a maximum of n eigenvectors, but these should include all of the variance. The B obtained this way is orthogonal, but not orthonormal, thus this is normalized to unit vectors.

This is implemented in python, and the code to calculate B in both the hard and the smart way, is shown in Listing 1 in Appendix. For the ORL database, $n = 200$ and $d = 112 * 92 = 10304$. The computation time to calculate B, using my personal computer, is 643.92 seconds when doing it the hard way (1.), compared to 0.79 seconds when doing it the smart way (2.). Thus, computationally, it is far more efficient to do it the smart way, when $n << d$. We will evaluate the performance of the two versions later.

After B is calculated, the training and test data can be transformed to a lower dimension space d'. This is done by calculating $X_{tr,d'} = X_{tr}B_{d'}$, where $B_{d'}$ is a matrix with the first d' columns of B (by inspection, the d' first eigenvectors corresponds to the d' highest eigenvalues, and thus they form the space with maximum variance). This is slightly different than in the class slides, where it says $Y = B^t_{d'}X$. Note however that since my X_{tr} is transposed compared to the one in the class slides, and since we have $(AB)^t = B^tA^t$, I could have calculated $X_{tr,d'} = (B^t_{d'}X^t_{tr})^t$, but it would be unnecessary when I want to end up with $X_{tr,d'}$ as a $n \times d'$ - matrix. Similarly, we calculate $X_{te,d'} = X_{te}B_{d'}$. Finally, to classify the test set, nearest neighbour is used on $X_{te,d'}$ with $X_{tr,d'}$ as training data. The nearest neighbour algorithm is shown in Listing 2 in Appendix.

The performance of eigenfaces on the ORL database is shown in figure 1 and 2.

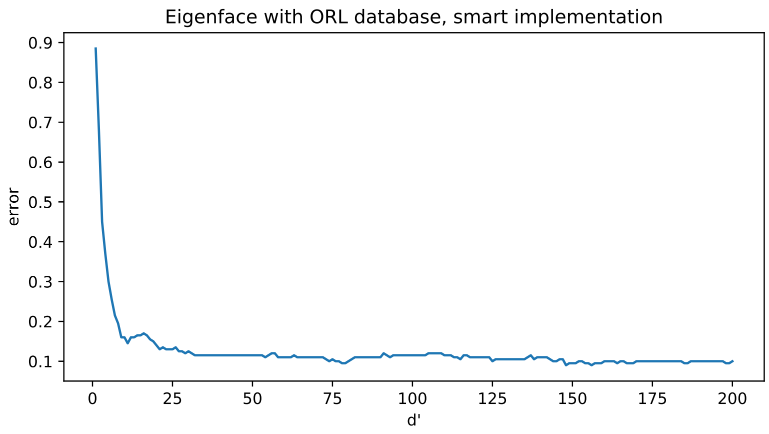

Figure 1: Error as a function of dimensions used in PCA, d', using the eigenface algorithm with the ORL database. This is the smart implementation described in (2.) above. Best error is 9 % with $d' = 148$.

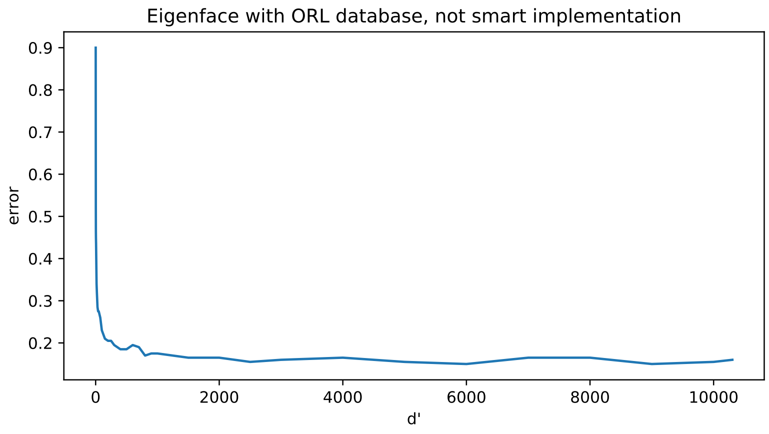

Figure 2: Error as a function of dimensions used in PCA, d', using the eigenface algorithm with the ORL database. This is the hard implementation described in (1.) above. Best error is 15 % with $d' = 6000$.

We see that the eigenface algorithm works well on the ORL database, with few errors. We also see that the smart implementation is performing better, in addition to it being faster. Thus, we can conclude that we should use the smart implementation when $n << d$. The best error is achieved using $d' = 148$ dimensions in the PCA analysis, but we also see from figure 1 that it improves little using more than $d' = 30$. The final choice of d' would depend on the importance of computational time, it is faster with fewer dimensions, but it is still very fast using $d' = 148$ dimensions. Over all, it seems to perform well.

The performance of eigenfaces on images from the Yale database is shown in figure 3.

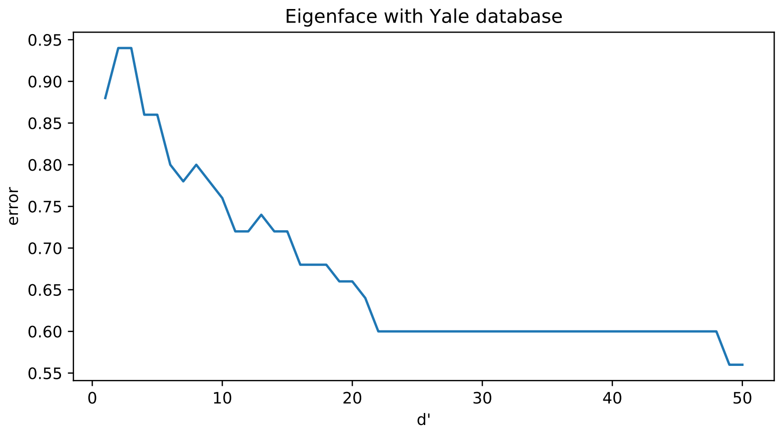

Figure 3: Error as a function of dimensions used in PCA, d', using the eigenface algorithm with a images from the Yale database with different light settings. This is the smart implementation described in (2.) above. Best error is 56 % with $d' = 49$.

We see from the figure that the eigenface algorithm is performing very bad with images from the Yale database. This is not really very surprising. PCA is a unsupervised method that transforms the data to a space where the variance is highest. When the light settings varies, we can expect the light to cause a higher variance than the between-class differences. Thus, the nearest neighbour is likely to be an image with similar light setting, instead of a similar face, also in the PCA-space.

2 Fisherfaces

The fisherface is implemented the following way: First the eigenface-algorithm is used to obtain $X_{tr,d'}$ and $X_{te,d'}$. This is because for the fisherface algorithm, there are some steps that will result in a singular matrix if the number of dimensions is higher than the number of training data, $d > n$. In that case, the algorithm will not work, thus it is necessary first to transform the train and test data to a reduced dimension space d'. Since there is no point in using more than $d' = n$ dimensions, we will always use the smart implementation in the eigenface algorithm. The next step is to calculate total mean μ_T and the within-class means $\mu_c, c = 1, ..., C$, from $X_{tr,d'}$. Then we calculate

$$S_b = \sum_{c=1}^{C} n_c(\mu_c - \mu_T)(\mu_c - \mu_T)^t \tag{1}$$

and

$$S_w = \sum_{c=1}^{C} \sum_{x \in c} (x - \mu_c)(x - \mu_c)^t, \tag{2}$$

where n_c is the number of samples belonging to class c, and x is an image from $X_{tr,d'}$. The goal of fisherface is to find B that maximizes $||B^t S_b B||/||B^t S_w B||$. The class slides shows that this is equivalent of solving $CB = B\Delta$, with $C = S_w^{-1} S_b$. Thus, first C is calculated, then the eigenvectors of C, B is found. The code to calculate the final B in fisherface is shown in Listing 3 in Appendix. After B is found, the training and testdata is transformed again so we get $X_{tr,LDA} = X_{tr,d'} B_{C-1}$ and $X_{te,LDA} = X_{te,d'} B_{C-1}$. Here, B_{C-1} are the $C - 1$ first columns of B. When B is calculated, it has a maximum of $C - 1$ corresponding eigenvalues that are non-zero, thus only the first $C-1$ eigenvectors take part in maximizing $||B^t S_b B||/||B^t S_w B||$. By inspection, the first $C - 1$ columns of B are the ones that corresponds to non-zero eigenvalues. Finally the nearest neighbour algorithm is used on $X_{te,LDA}$, with $X_{tr,LDA}$ as training data.

The performance of the fisherface algorithm is shown in figure 4 and 5.

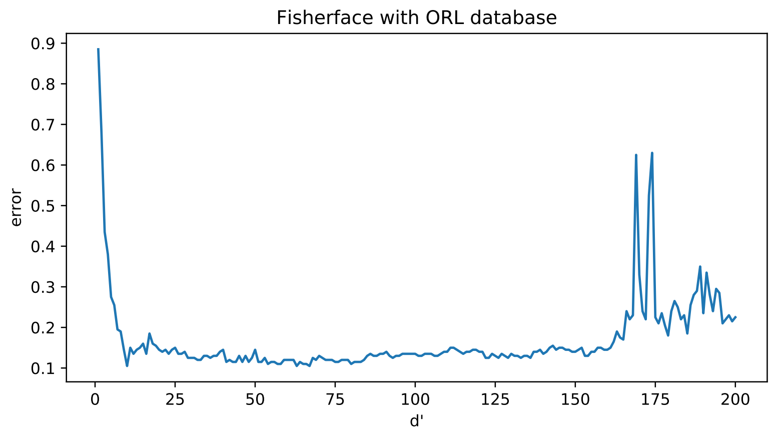

Figure 4: Error as a function of dimensions used in PCA, d', using the fisherface algorithm with the ORL database. Best error is 10.5 % with $d' = 10$.

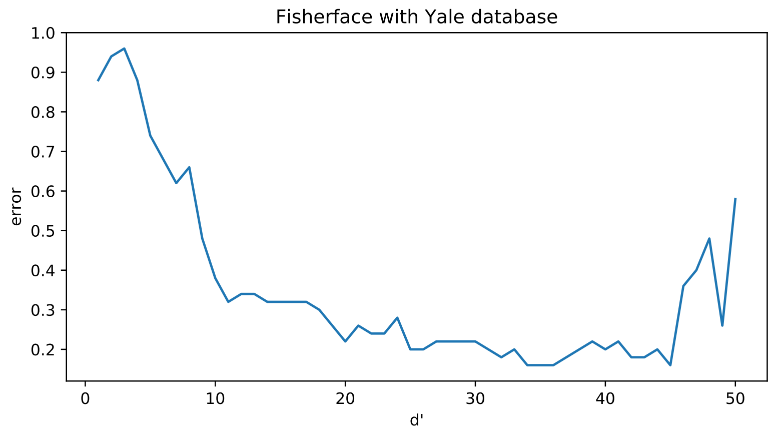

Figure 5: Error as a function of dimensions used in PCA, d', using the fisherface algorithm with images from the Yale database with different light settings. Best error is 16 % with $d' = 34$.

We see from the figures that the fisherface algorithm is quite similar in performance as the eigenface algorithm with the ORL database. In fact, the eigenface algorithm has the best error of the two, even though LDA is a supervised method. Since the ORL database has images with good light settings, the main variance picked up by the eigenface algorithm seems indeed to be the between-class variance, and thus it is possible for eigenface to outperform fisherface here. The eigenface algorithm was however very bad with the Yale database, and we expect fisherface to perform better here, since it is a supervised method and will find a space that maximizes the between-class variance, and minimizes the within-class variance, regardless of light settings. We see from the figure the performance of fisherface with the Yale images is far superior to the performance of eigenface on the Yale images. The fisherface algorithm takes a little more time due to having more computations to do, and it seems the motivation for using fisherface instead of eigenface is if the images has elements that might make the within-class variance high. We also see from both the figures that when d' approaches n, the error increases again. As mentioned, there will be problems with singular matrices if $d' > n$, and it seems this effect starts to influence the performance when d' is close to n.

3 Local light normalization

The local light normalization is implemented by first computing the image integral and the squared image integral for the image. Then, we let a window of size $k \times k$ slide over the image. For each window placement, the window sum S_1, and the squared window sum S_2 is calculated, using the pre-calculated image integrals. Then the window mean is calculated as $\mu_w = S1/k^2$. For the variance, we get the following:

$$Var(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i^2 - 2x\mu + \mu^2) = \frac{1}{n}(\sum_{i=1}^{n} x_i^2 - 2\mu\sum_{i=1}^{n} x_i + n\mu^2) = \frac{1}{n}(S_1 - S_2^2/n) \quad (3)$$

Thus, $\sigma^2 = \frac{1}{n}(S_1 - S_2^2/n)$. Then for each pixel in the window, $f(w(x,y)) = (w(x,y) - \mu_w)/\sigma_w$ is calculated and added to a new matrix. Finally, each pixel in the new matrix is divided by the number of window appearences for the pixel. The code to perform the local normalization is shown in Listing 4 in appendix. An image before and after the normalization with a window size of 10×10 is shown in figure 6.

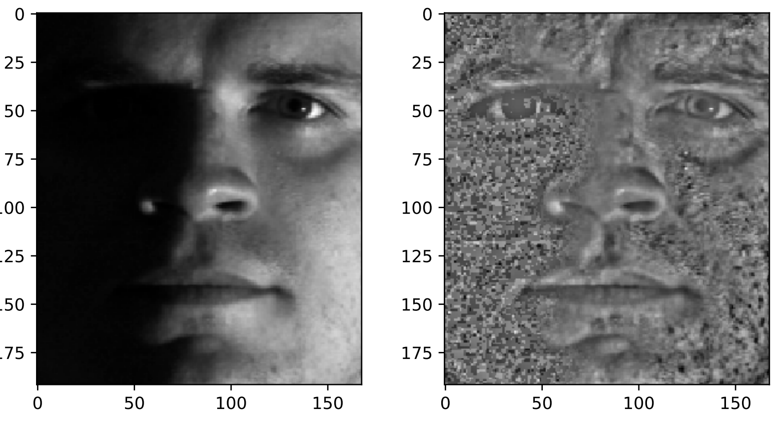

Figure 6: Image before and after light normalization, using a window of size 10×10.

The performance of eigenfaces and fisherfaces on the Yale images after using light normalization is shown in figure 7 and figure 8.

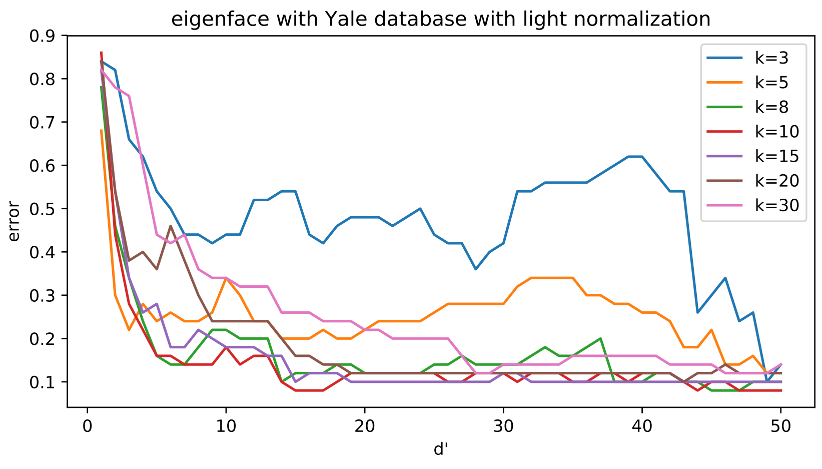

Figure 7: Plot showing error as a function of d' after doing light normalization with different window sizes k using the eigenface algorithm.

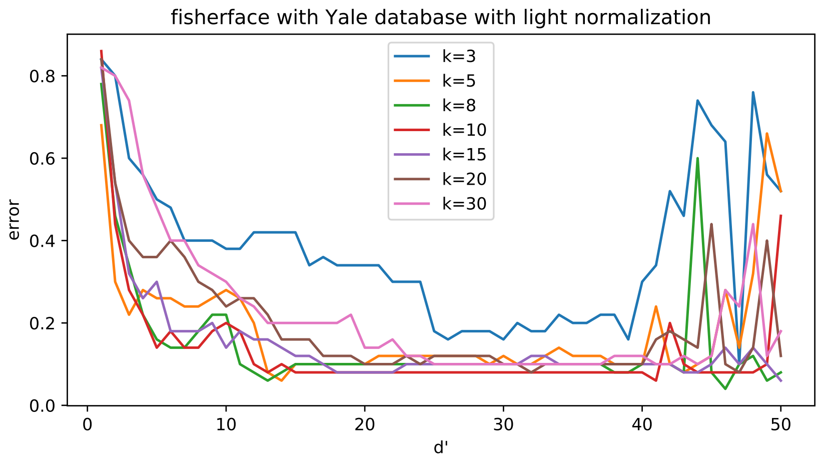

Figure 8: Plot showing error as a function of d' after doing light normalization with different window sizes k using the fisherface algorithm.

We see that the performance of the eigenface algorithm with the Yale images is drastically improved when using light normalization, with a minimum error of 8 %. It seems that when the light normalization is used, the main source of the variance becomes the between-class variance, so that the nearest neighbour algorithm works well in the transformed space. The performance of eigenfaces with the Yale images using light normalization is similar to the performance of eigenfaces with the ORL database. Thus it seems the light normalization works very well. The performance of the eigenface algorithm is also improved a lot when using light normalization, with a minimum error of 4 %. Thus, fisherface is also better than eigenface on the Yale images after light normalization. Thus it seems the supervised method works somewhat better after light normalization. It seems like a window size of $k = 10$ works well with both the algorithms.

Note that the Yale images are only a fraction of the entire database, and are picked in a random-ish way. There are quite few individuals, thus the overall distances between the individuals will become bigger, making the error rate lower for the Yale images compared to the ORL database, even when the ORL images are of far better quality. The tests I have performed are not good for comparing the performance on good light/bad light images, it merely shows that the different algorithms I have implemented work well, and shows how the different methods work together.

Appendix

Listing 1: Algorithm to calculate B for eigenfaces

```python
def getPC(trainData, smart=True, timer=False, save=False, load=False):

    # Possible to save and load the B-matrix since it takes a long time to calculate
    if load:
        eigVec = np.load("longPC.npy")
        return eigVec

    # To compare the computation time of the hard or smart way
    if timer:
        startTime = time.time()

    # Constants
    n=len(trainData)
    dim=len(trainData[0])

    # Calculates mu, A, and At. trainData seems to be transposed compared to the class slides,
        so
    # A/At is calculated different to obtain same size ad in class slides. A is d x n.
    trainDataMu = np.sum(trainData, axis=0) / n
    At = trainData - trainDataMu
    A = np.transpose(At)

    # Follows the steps described as the 2. alternative in report
    if smart:
        C = 1 / dim * np.matmul(At, A)
        eigVal, eigVec = np.linalg.eig(C)
        eigVec = np.matmul(A, eigVec)
        eigVec = normalize(eigVec, norm='l2', axis=0)

    # Follows the steps described as the 1. alternative in report
    else:
        C = 1/n*np.matmul(A,At)
        eigVal, eigVec = np.linalg.eig(C)

    # To compare the computation time of the hard or smart way
    if timer:
        endTime=time.time()
        print("Time of PCA: ", endTime-startTime)

    # Possible to save and load the B-matrix since it takes a long time to calculate
    if save:
        np.save("longPC.npy", eigVec)

    # eigVec is B in report.
    return eigVec
```

Listing 2: Nearest neighbur

```python
def nearestNeighbur(trainX, testX, trainY):

    predictionList = np.zeros(len(testX))

    # For each test sample, calculates the distance to all train samples, and predicts as the
        closest.
    for i in range(len(testX)):
        difference = np.sum((trainX - testX[i]) ** 2, axis=1)
        predictionList[i] = trainY[np.argmin(difference)]

    return predictionList
```

Listing 3: Algorithm for calculating B for fisherfaces

```
def getLDA(trainDataX, trainDataY):

    # Calculate total class mean
    totalMean = np.sum(trainDataX, axis=0)/len(trainDataX)

    #Calculate within-class means, as well as numbers of samples for each class
    classes = np.unique(trainDataY)
    classMean = np.zeros((len(classes), len(trainDataX[0])))
    numberClass = np.zeros(len(classes))
    for i in range(len(classes)):
        numberClass[i] = len(np.where(trainDataY==classes[i])[0])
        classMean[i] = np.sum(trainDataX[np.where(trainDataY==classes[i])],
            axis=0)/numberClass[i]

    # Calculates Sb and Sw
    Sb = np.zeros((len(trainDataX[0]), len(trainDataX[0])))
    Sw = np.zeros((len(trainDataX[0]), len(trainDataX[0])))

    for i in range(len(classes)):

        Sb += (numberClass[i]*np.outer(classMean[i]-totalMean, classMean[i]-totalMean))

        XMinusClassMean = trainDataX[np.where(trainDataY==classes[i])]-classMean[i]
        for j in XMinusClassMean:
            Sw += np.outer(j, j)

    # Calculate C
    C = np.matmul(np.linalg.inv(Sw), Sb)

    # Eigvec of C is B in fisherface
    eigVal, eigVec = np.linalg.eig(C)

    return eigVec
```

Listing 4: Local light normalization

```
def lightNormalize(image, reshape1, reshape2, size):

    # Reshape image to matrix, and make matrices to store image integrals
    image = image.reshape((reshape1,reshape2))
    sumImage = np.zeros((reshape1, reshape2))
    sumImageSquared = np.zeros((reshape1, reshape2))

    # Image integral and squared image integral
    sumImage[0][0] = image[0][0]
    sumImageSquared[0][0] = image[0][0] ** 2

    for i in range(1,reshape1):
        sumImage[i][0] = sumImage[i - 1][0] + image[i][0]
        sumImageSquared[i][0] = sumImageSquared[i - 1][0] + image[i][0] ** 2

    for i in range(1,reshape2):
        sumImage[0][i] = sumImage[0][i - 1] + image[0][i]
        sumImageSquared[0][i] = sumImageSquared[0][i - 1] + image[0][i] ** 2
```

```python

    for i in range(1,reshape1):
        for j in range(1,reshape2):
            sumImage[i][j] = sumImage[i-1][j]+sumImage[i][j-1]+image[i][j]-sumImage[i-1][j-1]
            sumImageSquared[i][j] = sumImageSquared[i - 1][j] + sumImageSquared[i][j - 1] + \
                                    image[i][j]**2 - sumImageSquared[i - 1][j - 1]

    # normalized stores f(w(x,y)), windowAppearences sums up appearences for each pixel, used
    # to average the pixels windows appearences in the end
    normalized = np.zeros((reshape1, reshape2))
    windowAppearences = np.zeros((reshape1, reshape2))

    #For each window position
    for i in range(reshape1-size+1):
        for j in range(reshape2-size+1):

            # Calculates the sum and squared sum of elements in window
            if j > 0 and i > 0:
                S1 = (sumImage[i+size-1][j+size-1] - sumImage[i-1][j+size-1] -
                    sumImage[i+size-1][j-1] + sumImage[i-1][j-1])
                S2 = (sumImageSquared[i+size-1][j + size-1] - sumImageSquared[i-1][j+size-1] -
                    sumImageSquared[i+size-1][j-1] + sumImageSquared[i-1][j-1])
            elif j > 0 and i == 0:
                S1 = (sumImage[i+size-1][j+size-1]-sumImage[i+size-1][j-1])
                S2 = (sumImageSquared[i+size-1][j+size-1]-sumImageSquared[i+size-1][j-1])
            elif j == 0 and i > 0:
                S1 = (sumImage[i+size-1][j+size-1] - sumImage[i-1][j+size-1])
                S2 = (sumImageSquared[i + size - 1][j + size - 1] -
                    sumImageSquared[i-1][j+size-1])
            else:
                S1 = sumImage[i+size-1][j+size-1]
                S2 = sumImageSquared[i + size - 1][j + size - 1]

            #Calculates n, window mean and window variance
            n=size**2
            mu = S1/n
            # Taking max of var and a small number, since the window variance in some positions
                with small
            # size seems to give zero variance.
            var = np.maximum(1/n * (S2 - (S1**2)/n), 1e-5)

            #Adds f(w(x,y)) to normalized[x,y], and adds one to each pixel appearing in window
            normalized[i:i+size, j:j+size] += (image[i:i+size, j:j+size]-mu)/np.sqrt(var)
            windowAppearences[i:i+size, j:j+size] += 1

    # Divides normalized[x,y] by number of appearences for pixel (x,y)
    normalized /= windowAppearences

    # Return as vector
    return normalized.reshape(-1)
```