Deep Learning For Computer Vision

Team Members:

Farah Ahmed T09	43-5680	farahelsamadony99@gmail.com
Karim Ebrahim T10	43-0414	karim.merhom@student.guc.edu.eg
Safa Magdy T10	43-2214	safa.ahmed@student.guc.edu.eg

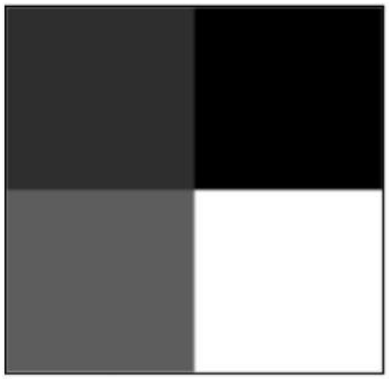
DR. Mohammed Abelmegeed Salem

Name of Teaching Assistant:

Eng.Mahmoud Mabrouk

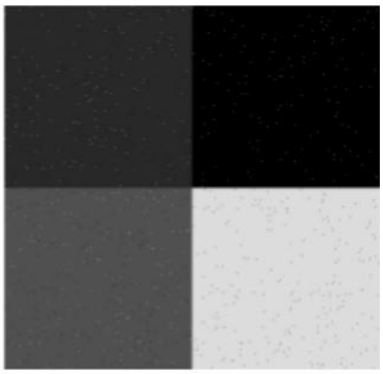
Task 1:

- To construct the image, We created an array with dimensions 512x512 as mentioned in the description.
- Then we divided the image into 4 quarters and assigned to each quarter a value then we save the image as a gray image.
- Then using salt and pepper noise we applied the noise to the original filter.
- Using a method called add_noise(img, x1,y1,x2,y2): takes the image and 4 number as an input.
- These numbers represent the range that will generate the number of noise pixels.
- For the light noise image, we passed small numbers to have a small range to produce a small number of noise pixels and so on in the other 2 images.

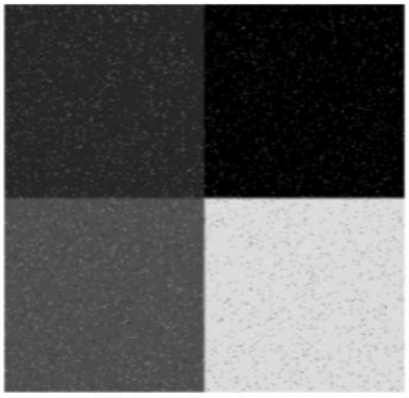


Original Image

Light noise Image



Medium noise Image



High noise Image

Task 2:

- The algorithm works by classifying the pixels according to the initial centers given and then calculating the means for the newly filled classes which will represent the new centers that will be used to reclassify the pixels.
- The previous step is repeated until convergence; where the centers no longer change.
- This algorithm was tested on 2 images from Dataset 3 (The original image that does not contain noise, and the medium noise image)
- As for the original image the accuracy was 100% and all the pixels were classified correctly because the pixel values were gathered at categories and the means can easily converge.
- The confusion matrix for the first image: [[65536, 0, 0, 0], [0, 65536, 0, 0], [0, 0, 65536, 0], [0, 0, 0, 65536]]
- As seen the number of pixels at each row are equal because all the pixels of the same color were correctly identified. The sum of all the rows will add up to the total number of pixels in the image (512x512).
- As for the second image that contains noise the accuracy was also 100%, and the reason for that is that the type of noise added was salt and pepper which can change the pixel value to either 0 or 255 which are two far numbers and will not affect the mean convergence.
- The confusion matrix for the second image: [[66061, 0, 0, 0], [0, 65180, 0, 0], [0, 0, 65164, 0], [0, 0, 0, 65739]]
- The number of pixels for each color is different since the categories are no longer equal in size and some pixels changed colors.
- The algorithm was also applied on 2 images from Dataset 2; a colored one and a gray-scale one, to test for segmentation ability.

Original Colored Image

Original Gray-scale Image

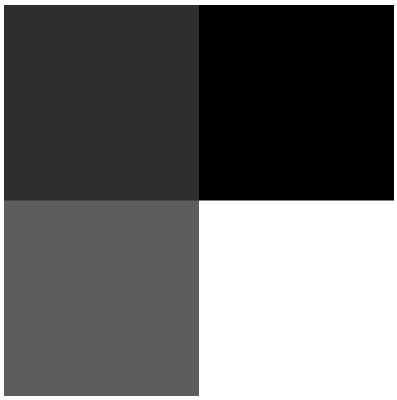
Segmented Colored k=4

Segmented Gray-scale k=2

- The colored image contained around 4 distinct colors, therefore we chose the number of classes k to be 4.
- As for the gray-scale image, we chose k=2 to segment the object from the background.

Task 3:

- The algorithm works to calculate the equations included in the EM algorithm.
- First we calculate the gaussian distribution of the data to be able to calculate the probability of each pixel to belong to each class.
- Then we recalculate the parameters in the algorithm and keep on repeating the algorithm until the difference between the new and old parameters reach a value that is less than the given e.
- The algorithm was tested on images of Dataset 3
- The first image is the one without noise: The accuracy for this image is 25%



```
classification_matrix, newParams, iterationCount, confusion_matrix = em(dataset3_0, 4,[1/1,1/2,1/3,1/4],
                                                                        [[25,1], [90, 1], [160,1], [240,1]], 0.01)
print(classification_matrix)
print(iterationCount)
print(confusion_matrix)
```

```
[3. 3. 3. ... 2. 2. 2.]
3
[[65536, 0, 0, 0], [0, 0, 0, 65536], [0, 65536, 0, 0], [0, 0, 65536, 0]]
```

- The second image is the Light noise: The accuracy is 25%

```
In [10]:
classification_matrix, newParams, iterationCount, confusion_matrix = em(dataset3_1, 4,[1/1,1/2,1/3,1/4],
                                                                        [[25,1], [90, 1], [160,1], [240,1]], 0.01)
print(classification_matrix)
print(iterationCount)
print(confusion_matrix)
```

```
[2. 2. 2. ... 1. 1. 1.]
3
[[65780, 0, 0, 0], [0, 0, 65437, 0], [0, 65445, 0, 0], [0, 65457, 25, 0]]
```

- The third image is the Medium Noise: The accuracy is 25%

```
In [18]:
classification_matrix, newParams, iterationCount, confusion_matrix = em(dataset3_2, 4,[1/1,1/2,1/3,1/4],
                                                                        [[25,1], [90, 1], [160,1], [240,1]], 0.05)
print(classification_matrix)
print(iterationCount)
print(confusion_matrix)
```

```
[2. 2. 2. ... 1. 1. 1.]
33
[[66061, 0, 0, 0], [0, 0, 65180, 0], [0, 65164, 0, 0], [0, 65168, 571, 0]]
```

- The fourth image is the High noise: The accuracy is 25%

```
In [20]: classification_matrix, newParams, iterationCount, confusion_matrix = em(dataset3_3, 4,[1/1,1/2,1/3,1/4],
                                                                                 [[25,1], [90, 1], [160,1], [240,1]], 0.04)
         print(classification_matrix)
         print(iterationCount)
         print(confusion_matrix)
```

```
[2. 2. 2. ... 1. 1. 1.]
48
[[67302, 0, 0, 0], [0, 0, 63711, 0], [0, 63627, 0, 0], [0, 63678, 3826, 0]]
```

- The following confusion matrices are resulting from using different values of e on the High Noise image.
- If e = 0.01: The accuracy is 25%, the number of iterations is 71

```
In [4]: classification_matrix, newParams, iterationCount, confusion_matrix = em(dataset3_3, 4,[1/1,1/2,1/3,1/4],
                                                                                [[25,1], [90, 1], [160,1], [240,1]], 0.01)
        print(classification_matrix)
        print(iterationCount)
        print(confusion_matrix)
```

```
[2. 2. 2. ... 1. 1. 1.]
71
[[67302, 0, 0, 0], [0, 0, 63711, 0], [0, 63627, 0, 0], [0, 63678, 3826, 0]]
```

- If e = 0.05: The accuracy is 25%, the number of iterations is 44

```
In [5]: classification_matrix, newParams, iterationCount, confusion_matrix = em(dataset3_3, 4,[1/1,1/2,1/3,1/4],
                                                                                [[25,1], [90, 1], [160,1], [240,1]], 0.05)
        print(classification_matrix)
        print(iterationCount)
        print(confusion_matrix)
```

```
[2. 2. 2. ... 1. 1. 1.]
44
[[67302, 0, 0, 0], [0, 0, 63711, 0], [0, 63627, 0, 0], [0, 63678, 3826, 0]]
```

- If e = 0.l: The accuracy is 25%, the number of iterations is 33.

```
In [6]: classification_matrix, newParams, iterationCount, confusion_matrix = em(dataset3_3, 4,[1/1,1/2,1/3,1/4],
                                                                                [[25,1], [90, 1], [160,1], [240,1]], 0.1)
        print(classification_matrix)
        print(iterationCount)
        print(confusion_matrix)
```

```
[2. 2. 2. ... 1. 1. 1.]
33
[[67302, 0, 0, 0], [0, 0, 63711, 0], [0, 63627, 0, 0], [0, 63678, 3826, 0]]
```

8. Increasing the level of noise will increase the number of iterations for the same value of e.

Increasing the value of epsilon will decrease the number of iterations because we are allowing for a higher difference value between the parameters.

The complexity of GMM will increase by increasing the data dimensionality because the GMM iterates over the pixels and when the dimensionality increases the number of iterations will increase and the computations will become more complex.

Task 4:

1. We made 2 Quantization methods one for the colored images and the other for the gray called QuantizationGrey and QuantizationColor.
2. In the methods, we used the mentioned equation to assign for each pixel the new value.
3. The only difference in the color images is that we have RGB so we had y1, y2, and y3.

Original Image

Quantized Image

Original Image

Quantized Image

Original Image

Quantized Image

Original image

Quantized image

Task 5:

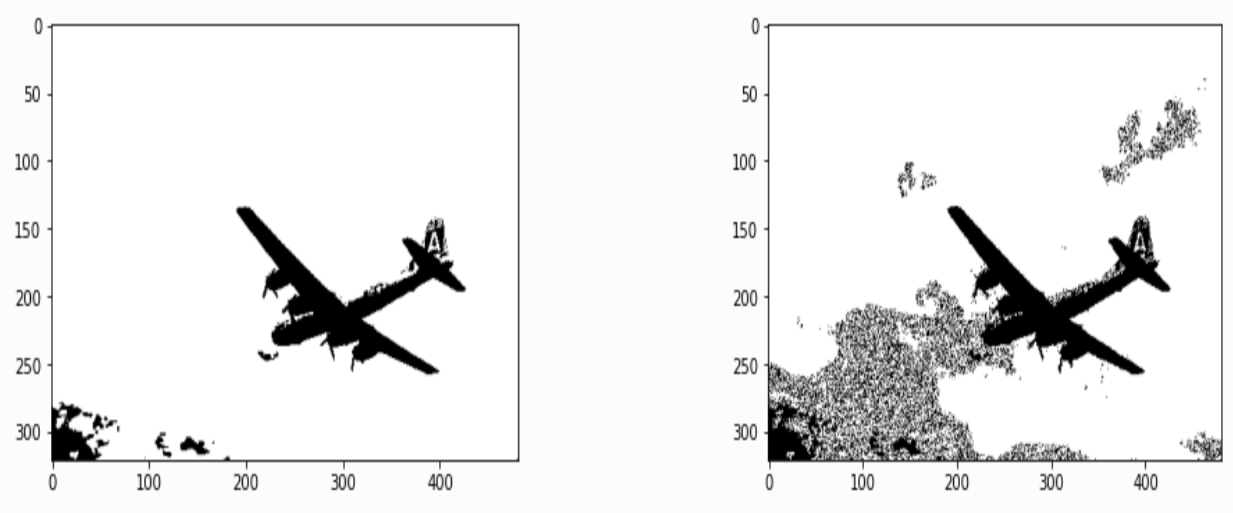
Naive Bayes Nearest Neighbor (NBNN) is a feature-based image classifier that achieves impressive degree of accuracy by exploiting ‘Image-to-Class’ distances and by avoiding quantization of local image descriptors.It’s performance is good as it avoid the vectorization step and instead use image to class comparison.

The Thresholding was done using Otsu Rule.

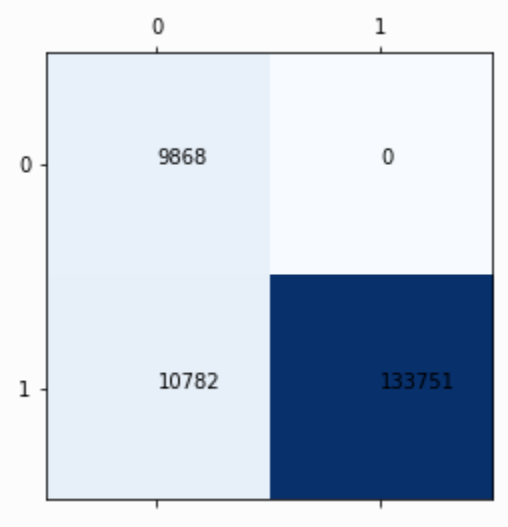
The first Img is the image after applying thresholding. And the second Img is after applying NaiveBayes and getting the predicted results

The image resulted from thresholding the object is segmented from the background

In the second Img the objects are segments from the background however there is some noise as Naive Bayes acts that each is the assumption of independent predictors.



The confusion Matrix between the predicted and the threshold.
The TrueZero(Zero and predicted Zero).
The FalseZero (Not Zero but predicted Zero)
The False 255(255 but predicted as a zero)
The True 255 (255 and predicted 255)



TrueZero =47051.
FalseZero=0.
False 255 = 2353
True 255 =104997