Ahmet Hakan Yılmaz

21803399

CS 464

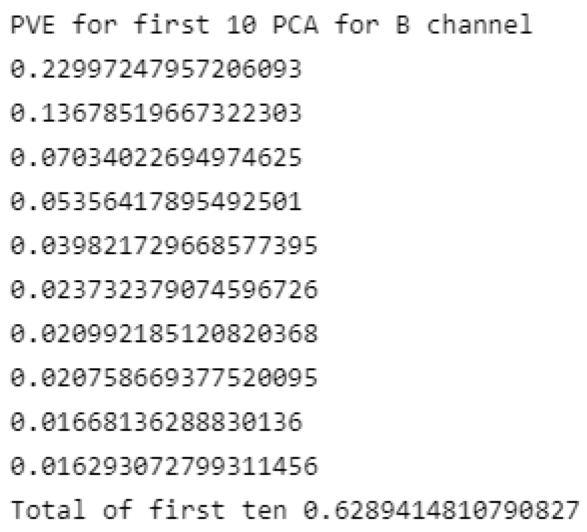
HW2

QUESTION 1

1.1

```
PVE for first 10 PCA for R channel
0.2150928295705477
0.13543590648775347
0.07504975573273255
0.051732173423000065
0.042289430973505064
0.024583277160715414
0.021772692745900598
0.01989857869202319
0.017070945760386022
0.016559603292283374
Total of first ten 0.6194851938388473
```

```
PVE for first 10 PCA for G channel
0.20045939393456677
0.13767809111140286
0.07695367428026664
0.053971234550888966
0.042918749347138435
0.02602259159431511
0.021426529166701794
0.02081297174873692
0.017393519667203514
0.016811649418805575
Total of first ten 0.6144484048200265
```



```
PVE for first 10 PCA for B channel
0.22997247957206093
0.13678519667322303
0.07034022694974625
0.05356417895492501
0.039821729668577395
0.023732379074596726
0.020992185120820368
0.020758669377520095
0.01668136288830136
0.016293072799311456
Total of first ten 0.6289414810790827
```

PVE is for the variance in data set that can show the contribution of each eigenvector to dataset. For R channel first 10 eigenvectors can explain the variance of 61.9 % in dataset, for G channel first 10 eigenvectors can explain the variance of 61.4 % in dataset, for B channel first 10 eigenvectors can explain the variance of 62.8 % in dataset.

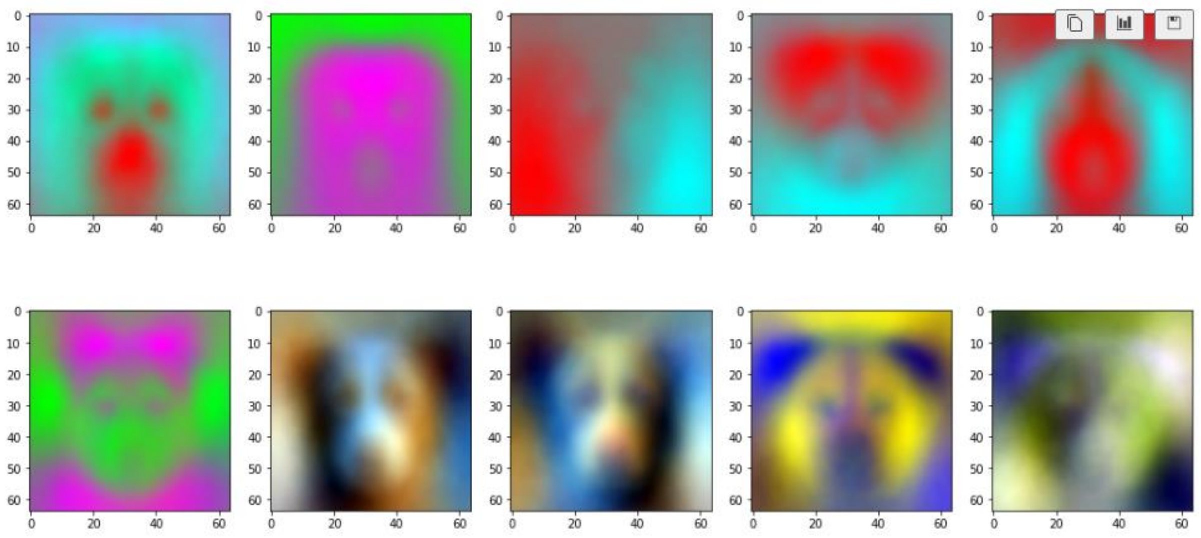
To pass 70% there are needed some more PCAs.

For channel R, to obtain %70 percent required number of PCA is: 18

For channel G, to obtain %70 percent required number of PCA is: 19

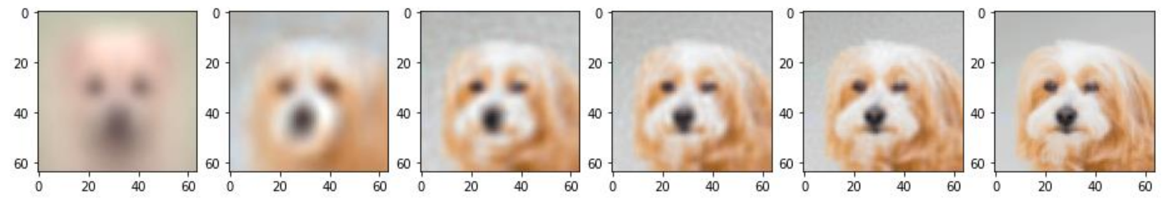
For channel B, to obtain %70 percent required number of PCA is: 17

1.2



We can see that all of the first 10 PCAs looks like a dog shadows. These are the representations of the eigenvectors we found and they explain the approximately 60 % of the variance in the dataset. (we found in part 1.1). At first it may come strange that colorfulness of those pictures however they are eigenvector representations derived from dataset and there were a lot of different coloured dogs in that dataset. And when different color channels eigenvectors combined it is normal to see that almost every color included in those eigenfaces.

1.3



We can get reconstructed image by applying matrix multiplication of eigenvectors with the vector of image which mean is subtracted. Then 1 more matrixmultiplication with transpose of eigen vectors provide as reconstructed images. I reconstructed the first image using first 1, 50, 250, 500, 1000, 4096 principal components like up. With principal components we get, we construct the image by showing the affect of those eigenvectors on that specific image. For example for k = 1, if we reconstruct an image we just uses 1 eigenvectors' contribution to that image and the image we get will be similar with most of the photos' first reconstructed images. As k increase the image we get images that is similar to the the original images. When k reaches 4096 we get the original image as there are already 4096 eigen vectors and all of them can explain every image in the dataset.As PVE of eigenvectors decrease the added eigenvectors affect on reconstruction will also decrease. That means for getting a very similar image to our original image we do not need to 4096 principle components.

QUESTION 2

In this question I took first 42000 row as training data and last 12000 as test data and rows between them as validation data.

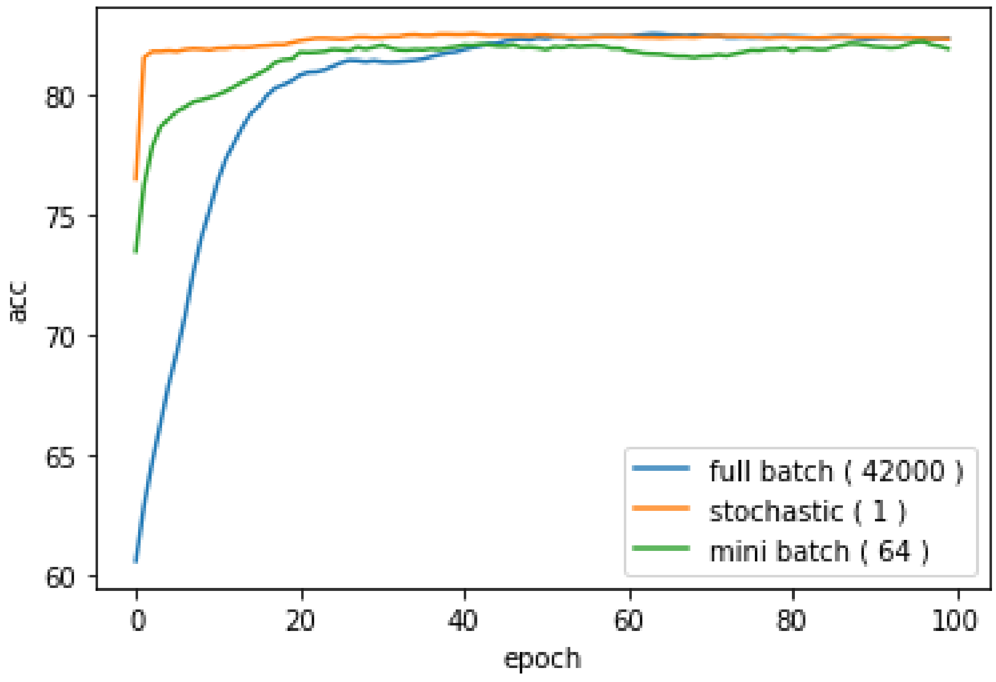
2.1

In question 2.1 I implemented all 3 batch sizes which are 1(stochastic gradient ascent) , 64 (mini-batch gradient ascent), and 42000 (full-batch gradient ascent). Iteration over 100 epochs, accuracy results were so much close to each other at the end. With a little difference full -batch gradient ascent is the one which has the most accuracy. Here are the prints for each 10 epoch and accuracy rates in validation set.

```
Batch Sıze 42000
epoch: 0 3636 6000 60.6
epoch: 10 4587 6000 76.45
epoch: 20 4851 6000 80.85
epoch: 30 4884 6000 81.4
epoch: 40 4915 6000 81.91666666666667
epoch: 50 4942 6000 82.36666666666666
epoch: 60 4947 6000 82.45
epoch: 70 4948 6000 82.46666666666667
epoch: 80 4947 6000 82.45
epoch: 90 4942 6000 82.36666666666666
Batch Sıze 1
epoch: 0 4592 6000 76.53333333333333
epoch: 10 4918 6000 81.96666666666667
epoch: 20 4937 6000 82.28333333333333
epoch: 30 4945 6000 82.41666666666667
epoch: 40 4952 6000 82.53333333333333
epoch: 50 4951 6000 82.51666666666667
epoch: 60 4947 6000 82.45
epoch: 70 4948 6000 82.46666666666667
epoch: 80 4944 6000 82.4
epoch: 90 4946 6000 82.43333333333334
Batch Sıze 64
epoch: 0 4411 6000 73.51666666666667
epoch: 10 4802 6000 80.03333333333333
epoch: 20 4908 6000 81.8
epoch: 30 4926 6000 82.1
epoch: 40 4926 6000 82.1
epoch: 50 4915 6000 81.91666666666667
epoch: 60 4911 6000 81.85
epoch: 70 4898 6000 81.63333333333334
epoch: 80 4911 6000 81.85
epoch: 90 4922 6000 82.03333333333333
```

```
Accuracy of batch size 42000: 0.8238333333333333
Accuracy of batch size 1: 0.8235
Accuracy of batch size 64: 0.8162350597609562
```

```
Confusion matrix of best model
(Full Batch)
TP: 1494 FP 660
FN: 397 TN 3449
```



It is expected that stochastic gradient starts and converges to its final value faster than other algorithms. It is normal that smaller batch sizes will start fast and converges to their final values faster as batch number can make model training computationally more expensive and slow. We can say that batch size affect training time as if batch size is bigger model learning happens in more epochs. There is one thing I did not expect is that I would expect bigger difference in final accuracy in favor of models with higher batch number.

2.2

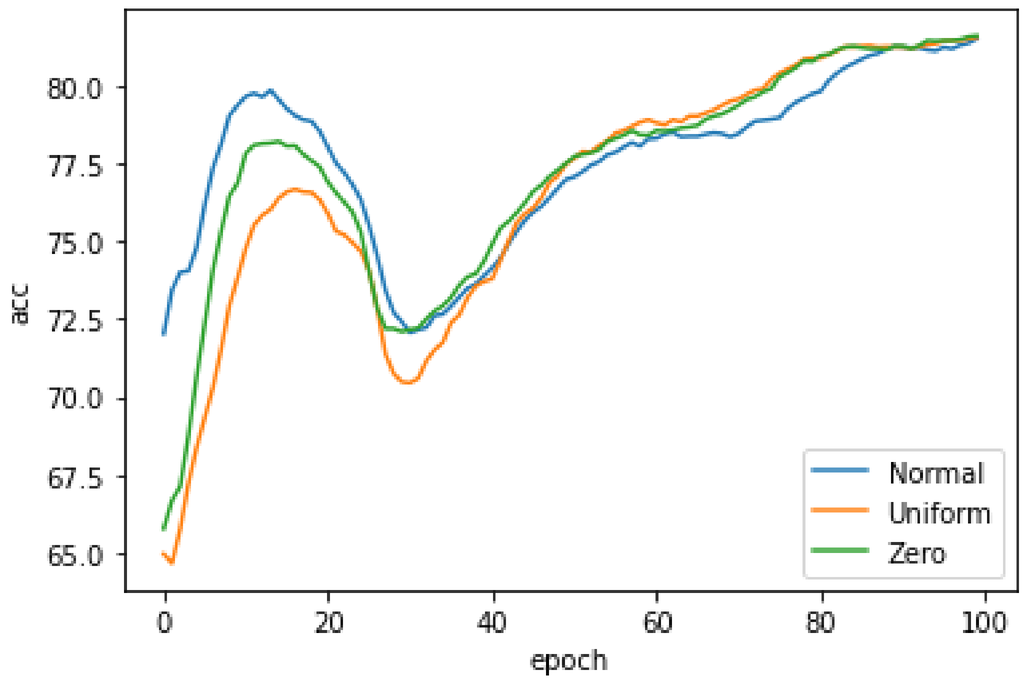
After trying all of 3 distributions I get the graph below for epochs and accuracy rates. In this section the important thing is that if the first weights we get is close to the most optimal weights (which we try to reach by updating weights) than our model will converges to its final value faster than other distributions. They are all very close to each other but the most accuracy at the and belongs to the zero distribution.

```
epoch: 0 4322 6000 72.03333333333333
epoch: 10 4780 6000 79.66666666666667
epoch: 20 4682 6000 78.03333333333333
epoch: 30 4324 6000 72.06666666666666
epoch: 40 4449 6000 74.15
epoch: 50 4625 6000 77.08333333333333
epoch: 60 4699 6000 78.31666666666666
```

```
epoch: 70 4708 6000 78.46666666666667
epoch: 80 4790 6000 79.83333333333333
epoch: 90 4875 6000 81.25
-----------------------------
epoch: 0 3897 6000 64.95
epoch: 10 4487 6000 74.78333333333333
epoch: 20 4554 6000 75.9
epoch: 30 4227 6000 70.45
epoch: 40 4428 6000 73.8
epoch: 50 4660 6000 77.66666666666667
epoch: 60 4729 6000 78.81666666666666
epoch: 70 4775 6000 79.58333333333333
epoch: 80 4854 6000 80.9
epoch: 90 4875 6000 81.25
-----------------------------
epoch: 0 3946 6000 65.76666666666667
epoch: 10 4670 6000 77.83333333333333
epoch: 20 4615 6000 76.91666666666667
epoch: 30 4330 6000 72.16666666666667
epoch: 40 4495 6000 74.91666666666667
epoch: 50 4664 6000 77.73333333333333
epoch: 60 4714 6000 78.56666666666666
epoch: 70 4763 6000 79.38333333333334
epoch: 80 4859 6000 80.98333333333333
epoch: 90 4877 6000 81.28333333333333
```

```
Normal: 0.8151666666666667
Uniform: 0.8156666666666667
Zeros: 0.8165749541437385
```

```
Confusion matrix of best model
(Zero)
TP: 1574 FP 580
FN: 523 TN 3323
```



2.3

I had found full batch was optimal in 2.1. So I implemented full batch gradient ascent with 4 different learning rate and I got the graph below. We can see that there are high fluctuations when learning rate is is 1. The reason behind that is that when updating our weights we can update so much than what is needed and it may cause parameters to move away from the optimal parameters we look for. However, surprisingly I also get the same attitude when learning rate is 0.00001. It is very surprising to get that. I believe it is an unusual situation and may be related to the random distribution and result of it we get. At the end learning rate with most accuracy is also 0.00001. I also had expected a middle variable 0.001 or 0.0001 as optimal variable with most accuracy as bigger and smaller learning rate can cause moving away from optimal or not getting fast enoug to optimal weights.

```
epoch: 0 3353 6000 55.88333333333333
epoch: 10 4732 6000 78.86666666666666
epoch: 20 4839 6000 80.65
epoch: 30 4843 6000 80.71666666666667
epoch: 40 3049 6000 50.81666666666667
epoch: 50 4628 6000 77.13333333333334
epoch: 60 4902 6000 81.7
epoch: 70 3911 6000 65.18333333333334
epoch: 80 3730 6000 62.166666666666664
epoch: 90 4912 6000 81.86666666666666

epoch: 0 3578 6000 59.63333333333333
epoch: 10 3624 6000 60.4
epoch: 20 3663 6000 61.05
epoch: 30 3705 6000 61.75
epoch: 40 3734 6000 62.233333333333334
epoch: 50 3767 6000 62.78333333333333
```

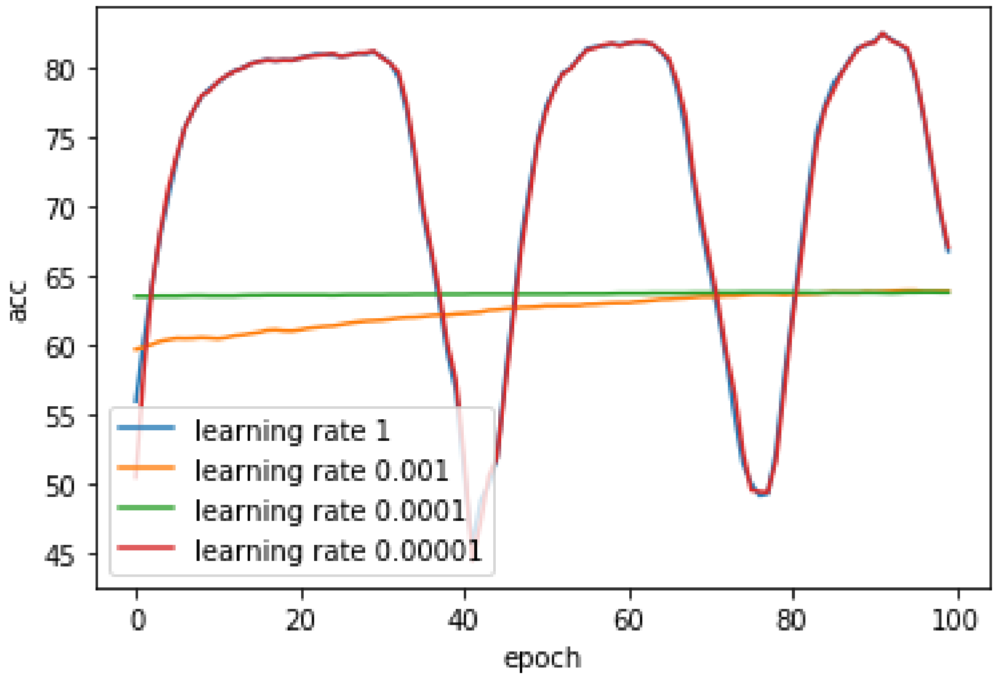
```
epoch: 60 3780 6000 63.0
epoch: 70 3807 6000 63.45
epoch: 80 3817 6000 63.61666666666667
epoch: 90 3826 6000 63.766666666666666

epoch: 0 3808 6000 63.46666666666667
epoch: 10 3810 6000 63.5
epoch: 20 3814 6000 63.56666666666667
epoch: 30 3815 6000 63.583333333333336
epoch: 40 3817 6000 63.61666666666667
epoch: 50 3818 6000 63.63333333333333
epoch: 60 3821 6000 63.68333333333333
epoch: 70 3823 6000 63.71666666666667
epoch: 80 3825 6000 63.75
epoch: 90 3824 6000 63.733333333333334

epoch: 0 3025 6000 50.416666666666664
epoch: 10 4736 6000 78.93333333333334
epoch: 20 4837 6000 80.61666666666666
epoch: 30 4839 6000 80.65
epoch: 40 3113 6000 51.88333333333333
epoch: 50 4609 6000 76.81666666666666
epoch: 60 4903 6000 81.71666666666667
epoch: 70 3954 6000 65.9
epoch: 80 3706 6000 61.766666666666666
epoch: 90 4904 6000 81.73333333333333


optimal Learn rate is 10 to the power  -5

----------
TP: 640 FP 1514
FN: 466 TN 3380
```



2.4

At question 2.4 after finding optimal hyperparameters on the validation set it is time to find how our model do against the test set. According to prior parts my optimal model uses full batch gradient ascent, its learning rate equals to 0.00001 and first distribution of our weights are zeros. When implementing that on test set we get results as

```
epoch: 0 3846 6000 64.1
epoch: 10 3846 6000 64.1
epoch: 20 3846 6000 64.1
epoch: 30 3848 6000 64.13333333333334
epoch: 40 3876 6000 64.6
epoch: 50 3937 6000 65.61666666666666
epoch: 60 4026 6000 67.1
epoch: 70 4126 6000 68.76666666666667
epoch: 80 4226 6000 70.43333333333334
epoch: 90 4295 6000 71.58333333333333
TP: 1077 FP 3272
FN: 38 TN 7613
Acc: 72.41666666666667
```

Precision is TP/ (TP +FP), recall is TP/(TP+FN), false positive rate FP / (FP +TN)

We know that Fbeta = ((1 + beta^2) * Precision * Recall) / (beta^2 * Precision + Recall)

Our metrics are

```
Precision 0.24764313635318463
Recall 0.9659192825112107
Accuracy 0.7241666666666666
F2 0.6113066182313542
F1 0.3274053807569539
F0.5 0.290908108692129
False positive rate 0.30059715204409737
```

If we consider that our task was to correctly classify whether a smart grid is unstable then if the rate of neg predictive value (FN/ (FN + TN)) is small than our model more accurately predicts when it predicst a grid as unstable. If false pos rate is small than our model more accurately predicts when the smart grid is actually unstable.

No it can change according to context of data. For example, in disease classification it is much more important to have a lower error while giving the results of actual diseased people. In our example it is also like that, lower percantage on unstable grids classification are more important than lower percantage on stable grids classification.