Assignment 2

Anita Bahmanyar

November 14, 2016

Question 3

3.1

3.2

This part of the assignment is asking for running the neural network code with different values of hyper parameters. The first part is to fix all the hyper parameters and change epsilon values from 0.001 to 1.0 for 5 different epsilon values. Figure 1 shows the cross entropy and accuracy of validation set for all 5 different epsilon values which I chose to be 0.001, 0.01, 0.1, 0.5 and 1.0.

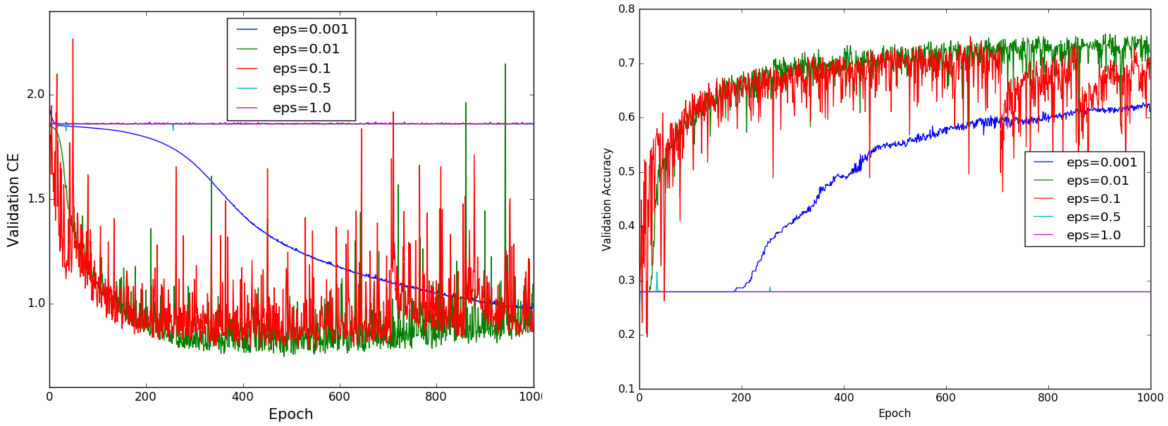

Figure 1:

What we conclude is that if epsilon is very small do instance in the case of 0.001, it takes very long for the validation to converge which means longer computation time. Also, if epsilon value is very large, for instance 0.5 and 1.0, then the accuracy curve would not be improving and it stays around 0.3 since we take large steps and we miss the global optimum value. I found that epsilon=0.01 and epsilon=0.1 have similar values and since the initialization is random, it is hard to say which is better since the accuracy of using these values are pretty similar and for each run they differ due to the randomness. The value of epsilon I choose is **epsilon=0.01** to do the other runs and I keep it 0.01 which changing other parameters.

In Figures 2 and 4 I show few different plots that are for the same epsilon values but they are separate plots for each epsilon so that we can see the training curve as well.

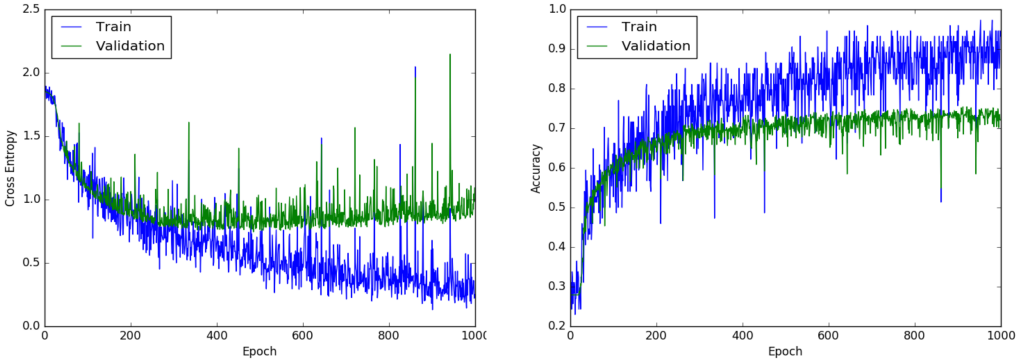

Figure 2: **epsilon=0.01**, CE on the left and accuracy on the right for both validation and training sets.

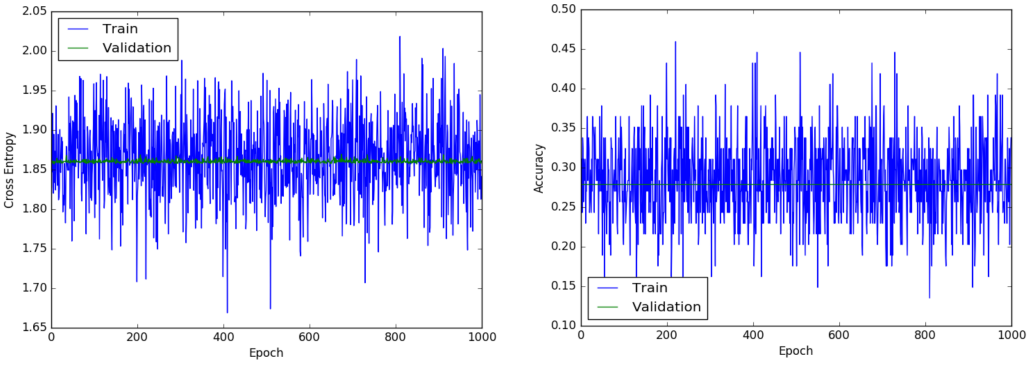

Figure 3: **epsilon=1.0**, CE on the left and accuracy on the right for both validation and training sets.

We also see that the accuracy of training set is always better than validation set which makes sense since we are training the model based on the training set. This is also true for cross-entropy where the training set always has smaller CE compared to the validation set.

In this part, I keep epsilon the same (0.01) and only change momentum value to be 0.0, 0.5 and 0.9. As it is shown in Figures **??** below, we see that momentum=0.9 does not return good results and we conclude it is too large. The results of momentum=0.0 and momentum=0.5 are similar except for the cross-entropy where for the case of momentum=0.5, the validation cross-entropy stars to go up at a much earlier time compared to momentum=0.0. When the validation CE starts increasing instead of decreasing that is the point where training further does not help which means it does not matter if we train more except that we spend more computational time, so the momentum value that I think is better in this case is **momentum=0.0** and I will use this value for the next part of the question.

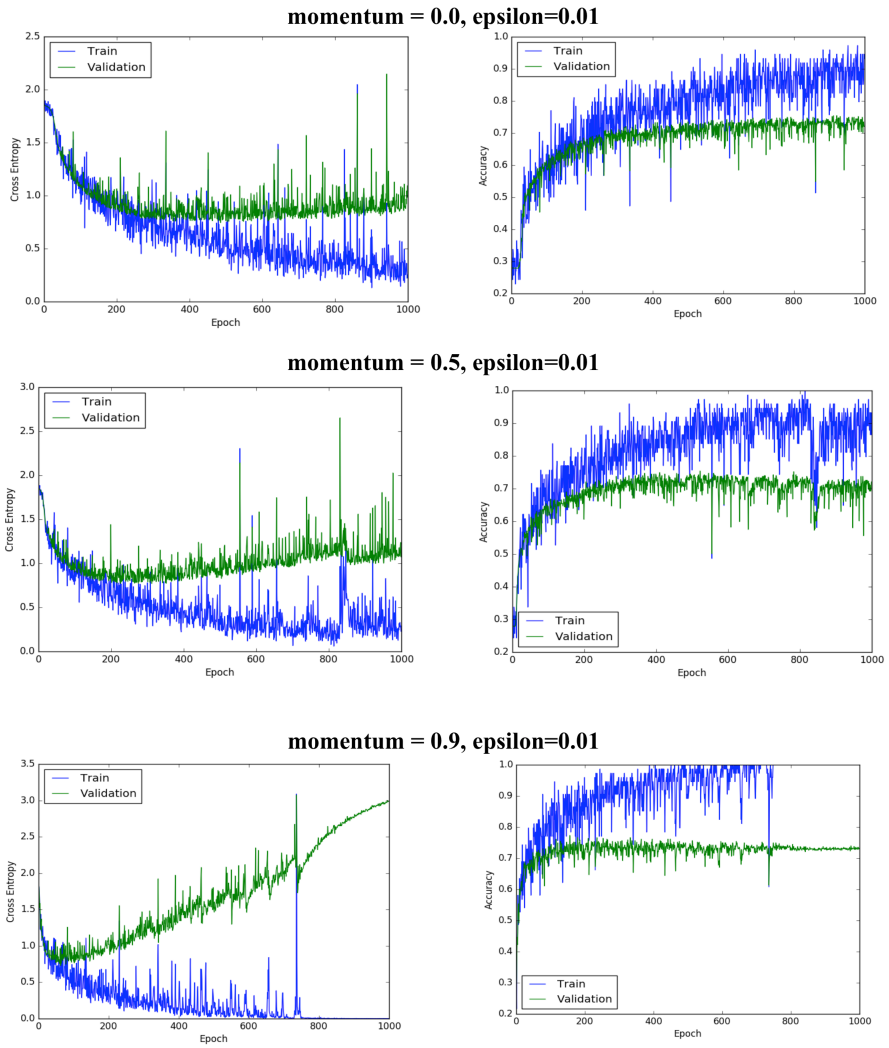

Figure 4: **epsilon=1.0**, CE on the left and accuracy on the right for both validation and training sets for three different momentum values while keeping **epsilon=0.01**. The momentum values are 0.0, 0.5 and 0.9 from top to the bottom.

Smaller batch sizes take longer to run due to having more steps to run since it is defined in the code that

```
num_steps = int(np.ceil(num_train_cases / batch_size))
```

so smaller batch size means more steps.

Q32_batch11.png

Figure 5: **epsilon=0.01, momentum=0.0, batch size=1**

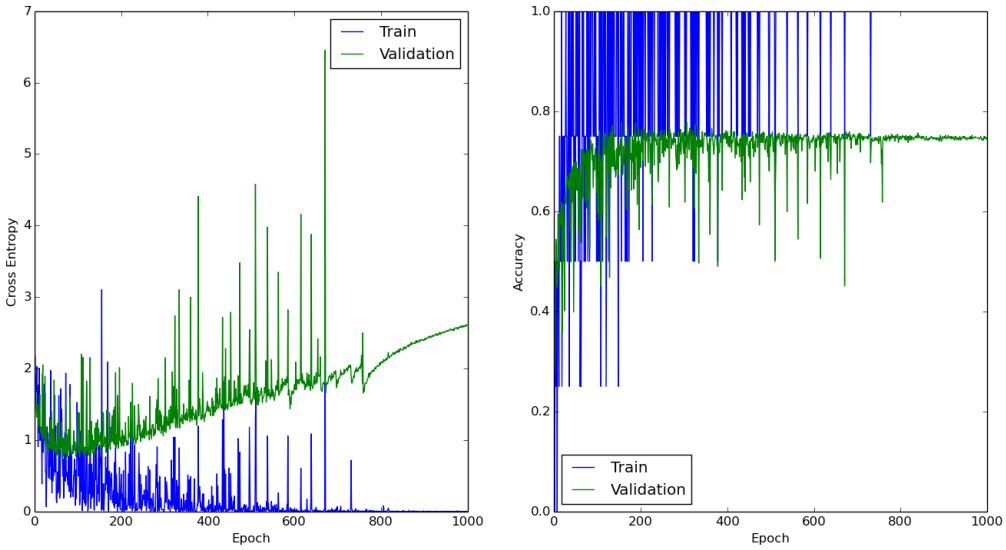

Figure 6: **epsilon=0.01, momentum=0.0, batch size=10**

Q32_batch100.png

Figure 7: **epsilon=0.01, momentum=0.0, batch size=100**

Q32_batch500.png

Figure 8: **epsilon=0.01, momentum=0.0, batch size=500**

Q32_batch1000.png

Figure 9: **epsilon=0.01, momentum=0.0, batch size=1000**

3.3

3.4

3.5

Question 4

4.2

In the code provided (mogEM.py), inside function mogEM there are a few lines that include "randConst" value.

```
p  = randConst + np.random.rand(K, 1)
p  = p / np.sum(p)   # mixing coefficients
mu = mn + np.random.randn(N, K) * (np.sqrt(vr) / randConst)
```

The first line will shows π_k values which are the mixing coefficients in the Gaussian mixture method. The mixing coefficients will be dominated by random values if randConst is small and it will be dominated by the value of randConst if it is large. In the third line, as we increase randConst, "(np.sqrt(vr) / randConst)" decreases and as we decrease the randConst value, this part of "mu" expression increases. This means that as we increase the randConst value, the "mu" values will be dominated by the mean values since the second part of "mu" expression will be very small.

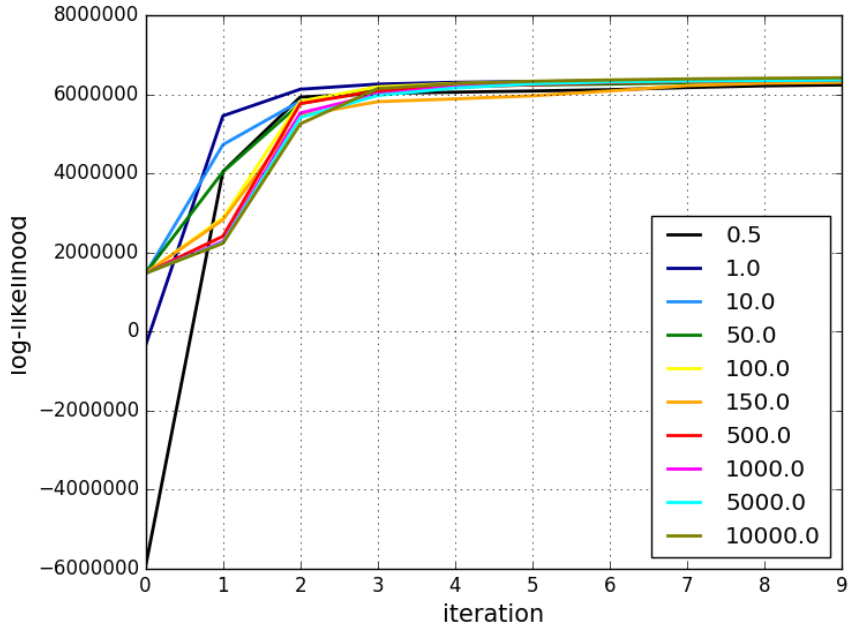

Figure 10: Log-likelihood as a function of number of iterations for different values of randConst parameter.

As we see in Figure 6, randConst=1.0 converges faster to the similar log-likelihood values compared to the other randConst values. So the model I choose has randConst=1.0. Below are variance and mean of the images shown for randConst = 1.0:

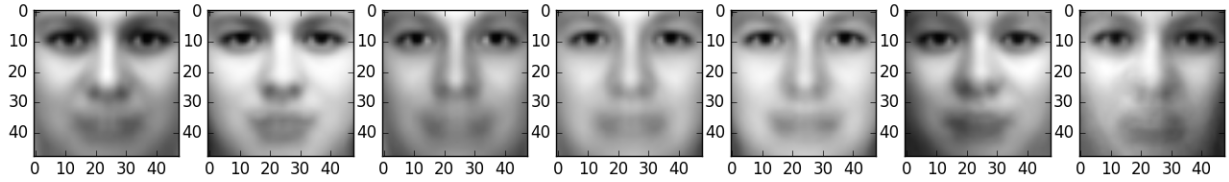

Figure 11: Mean of the images- Is is blurry because it is average.

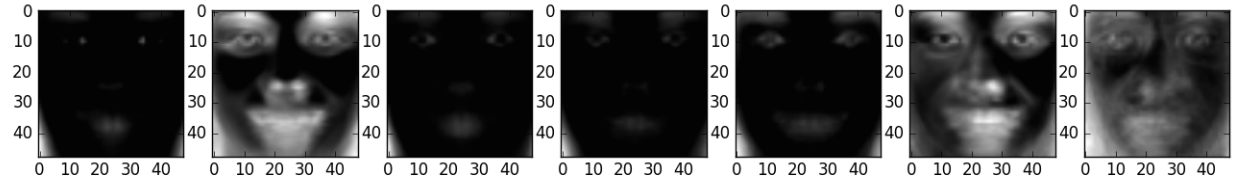

Figure 12: Variance of the images-black means lower variance and white means higher variance.

We see that most of these images look the same to us even though they are not and this is because there are small variations in the images. So variances are the most helpful ones in this case to see the variation between images. Also, black in the variance images mean lower variance and whiter means higher variance. Areas with lower variance are better so it is better if we see a lot of black areas in the variance images. Lower variance means better classifications so the variance images with more black area are better classifiers.

Below in Figure 9, the values of Gaussian mixture coefficients are given vs. cluster number.

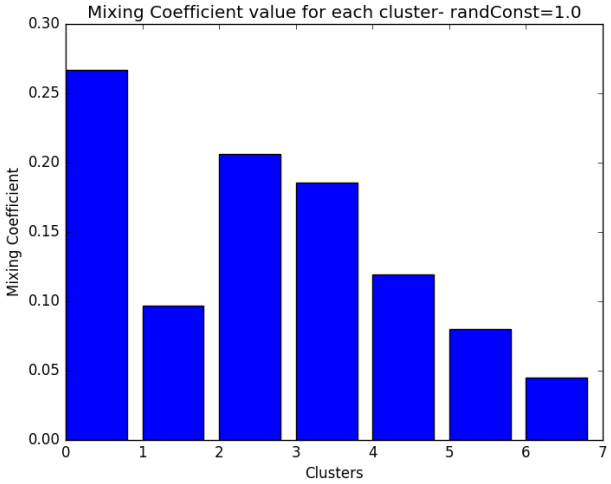

Figure 13: Value of mixture of coefficients for each cluster.

4.3

Initializing the means using K-means makes the code converge much faster than just using random values for the means. This can be seen in Figure 12 below.

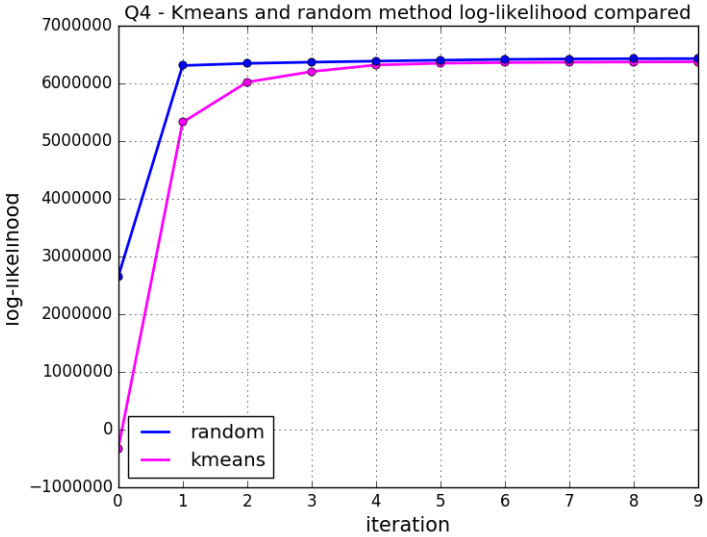

Figure 14: Comparison between log-likelihood using two different methods: Kmeans shown in magenta and randomized method (using randConst) shown in blue. We can see that using means method the log-likelihood converges earlier to a similar value compared to the randomized method.

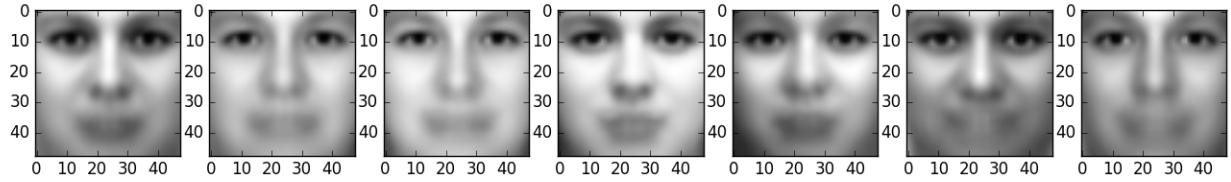

Figure 15:

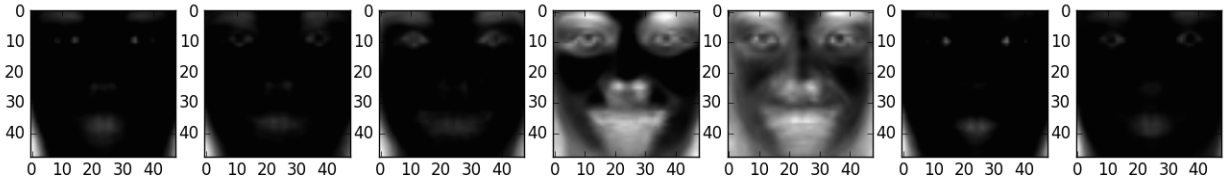

Figure 16:

4.4

In this question, we are trying to compute $p(d|x)$ value using Baye's rule. using Baye's rule we know that:

$$p(d|x) = \frac{p(x|d)p(d)}{p(x)}, \tag{1}$$

where $p(x|d)$, $p(d)$ is the prior and $p(x)$ is the evidence. Since $p(x)$ is constant for all the values we can ignore it here. Then, we can take log of both sides of Equation (1) and write it as:

$$\log(p(d|x)) = \log(p(x|d)p(d)) = \log(p(x|d)) + \log(p(d)). \tag{2}$$

We do have value of $p(d)$ from "log_likelihood_class" function in the code provided and value of $p(x|d)$ is given in the function called "mogLogLikelihood". So we can use these functions to compute $\log p(d|x)$.

Figure 14 shows the for all three cases of training set, validation set and test set.

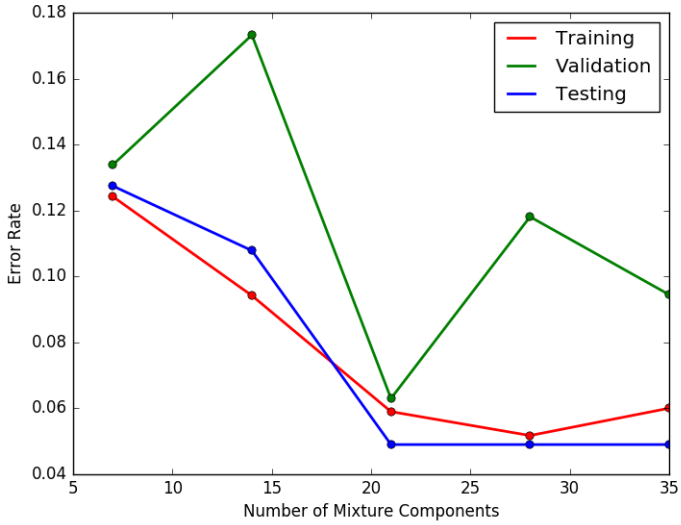

Figure 17:

Just to see the results because we have some randomness in the values we start with, I ran this part of the code again and the result is shown below for the same values as Figure 13. Figure 14 shows for all three cases of training set, validation set and test set.

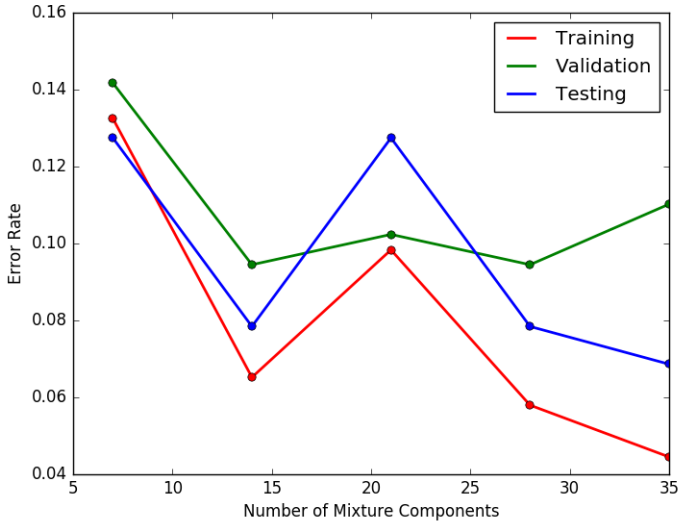

Figure 18: