Assignment 2

Anita Bahmanyar

November 14, 2016

Question 3

3.1

We want to show the result of the runs for both NN and CNN methods with the default parameter values. The figures are shown below:

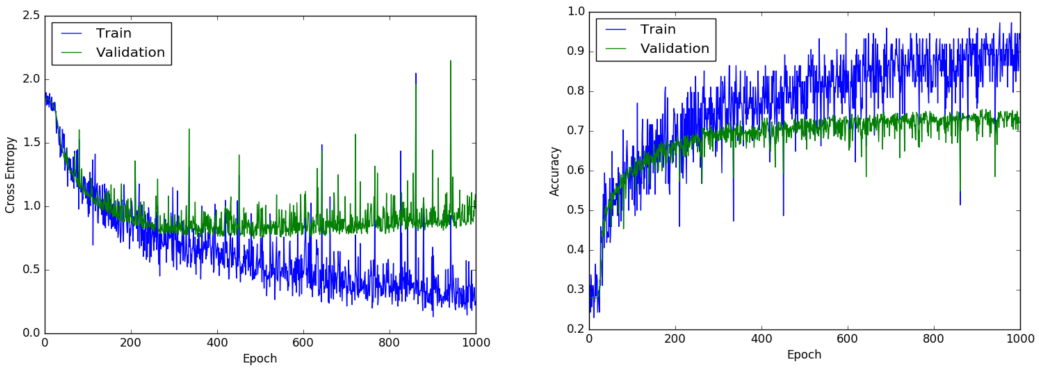

Figure 1: NN method used, eps=0.01, momentum=0.0, batch size=100

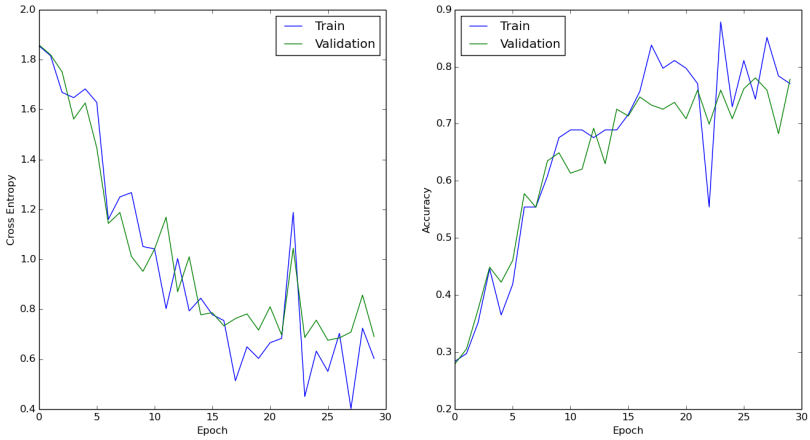

Figure 2: CNN method used, eps=0.1, momentum=0.0, batch size=100

As it is shown in the figures, validation set accuracy is always smaller than the training set which makes sense since we are training the model based on the training set and not on the validation set. As a result, the cross-entropy of validation set is also always larger than that of the training set meaning the model works better for the training set than the validation (since smaller cross-entropy means better results).

3.2

NN

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 3 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.

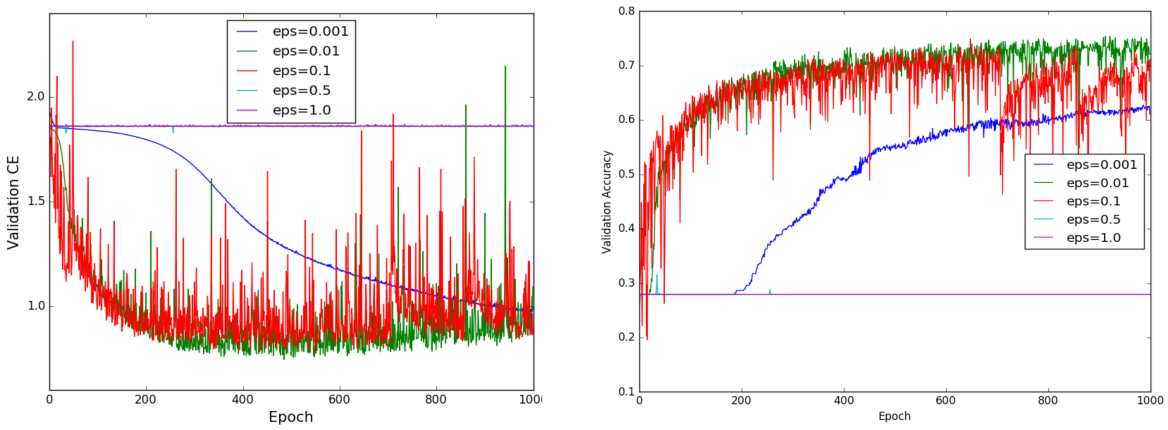

Figure 3:

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 4 and 6 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.

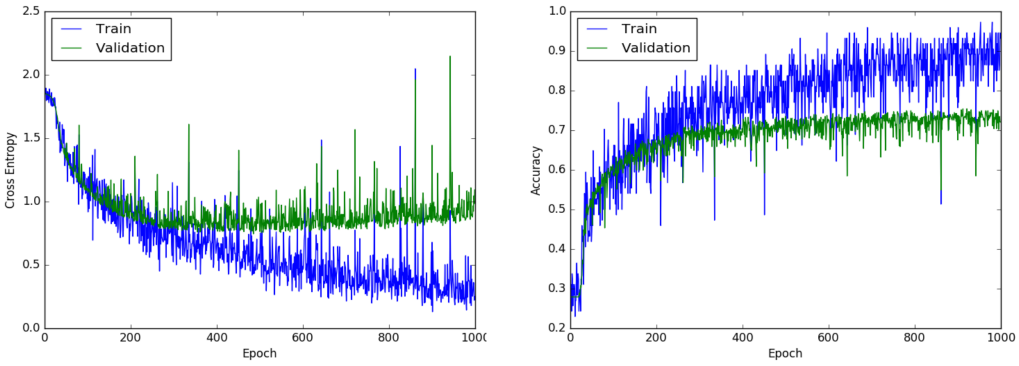

Figure 4: **epsilon=0.01**, CE on the left and accuracy on the right for both validation and training sets.

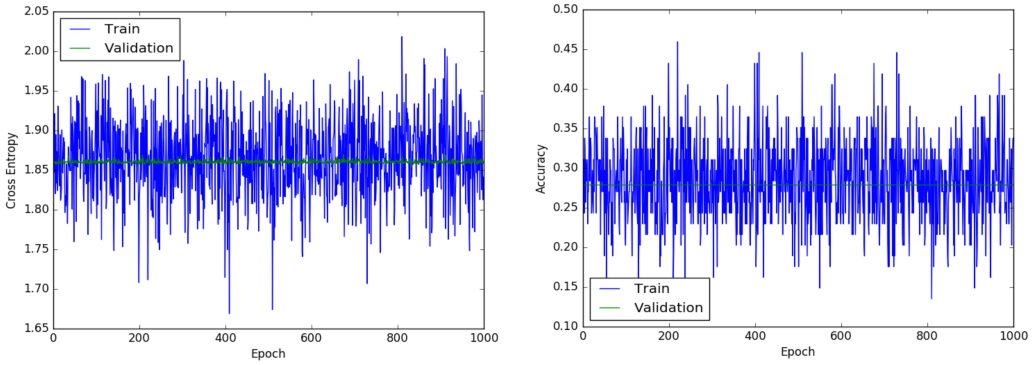

Figure 5: **epsilon=1.0**, CE on the left and accuracy on the right for both validation and training sets.

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.

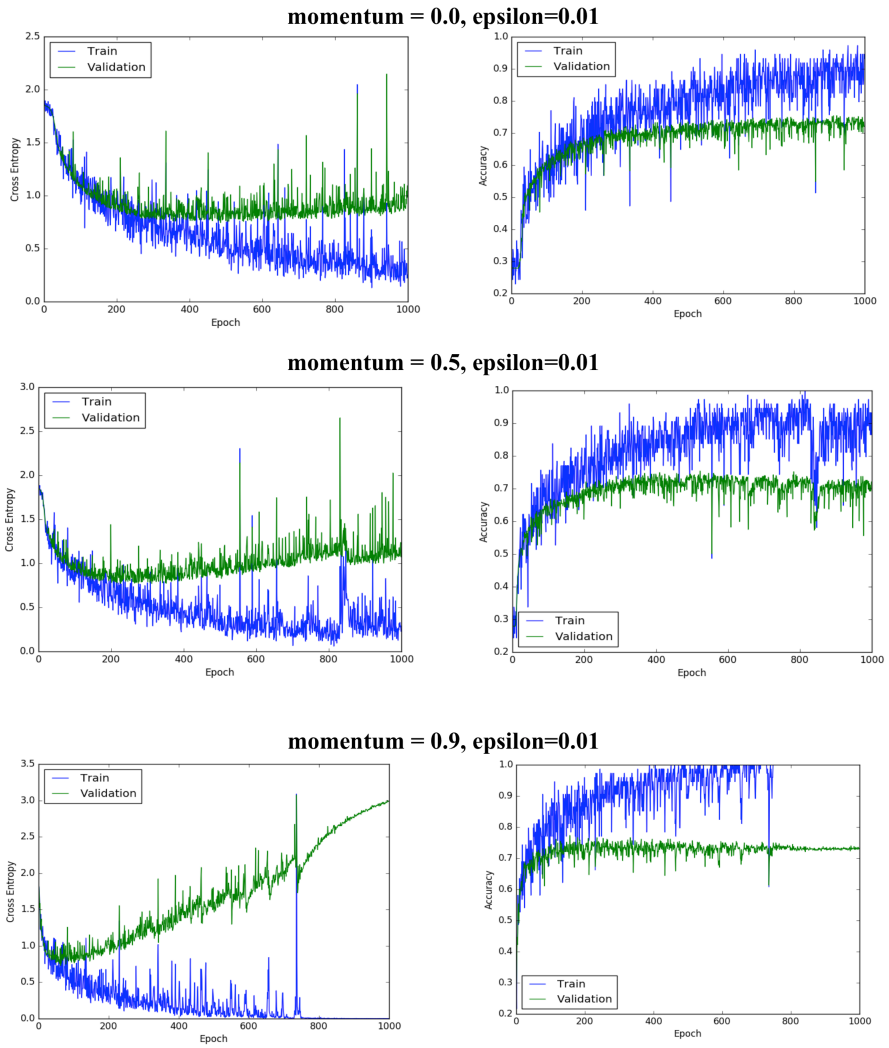

Figure 6: **epsilon=1.0**, CE on the left and accuracy on the right for both validation and training sets for three different momentum values while keeping **epsilon=0.01**. The momentum values are 0.0, 0.5 and 0.9 from top to the bottom.

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.

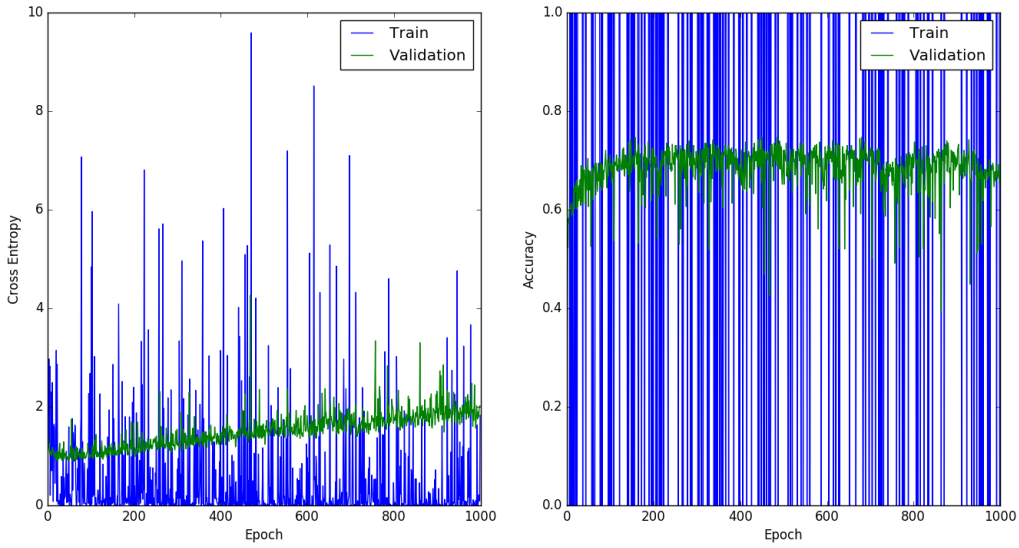

Figure 7: **epsilon=0.01, momentum=0.0, batch size=1**

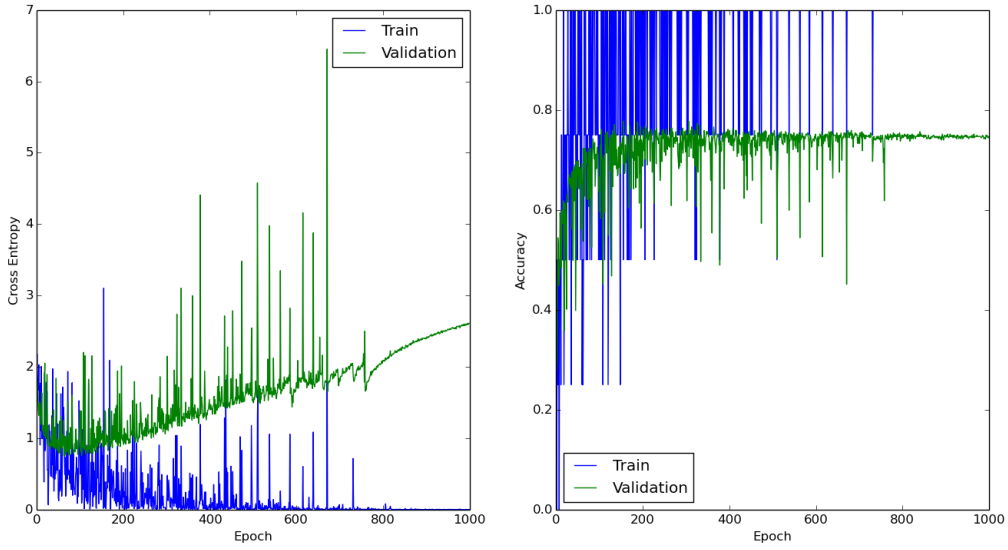

Figure 8: **epsilon=0.01, momentum=0.0, batch size=10**

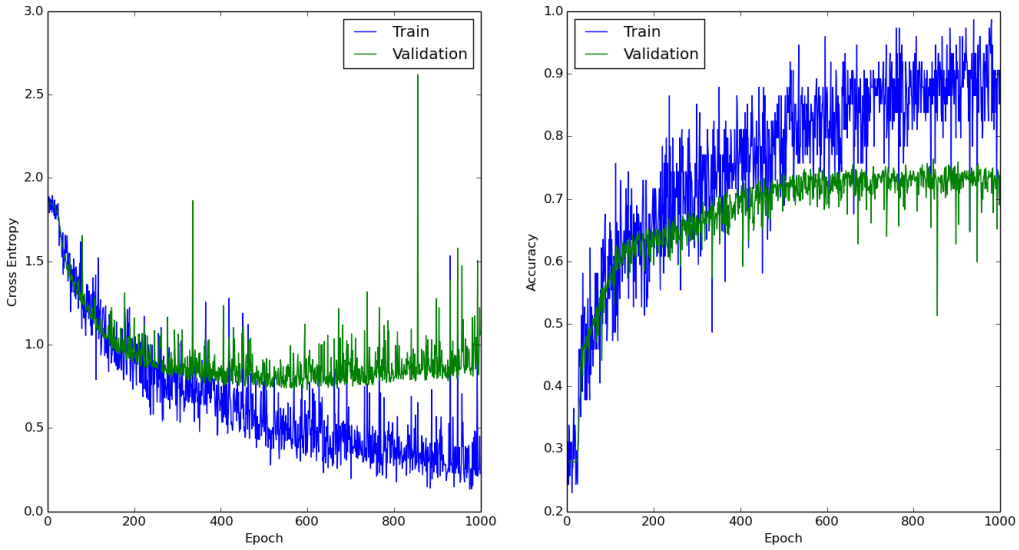

Figure 9: **epsilon=0.01, momentum=0.0, batch size=100**

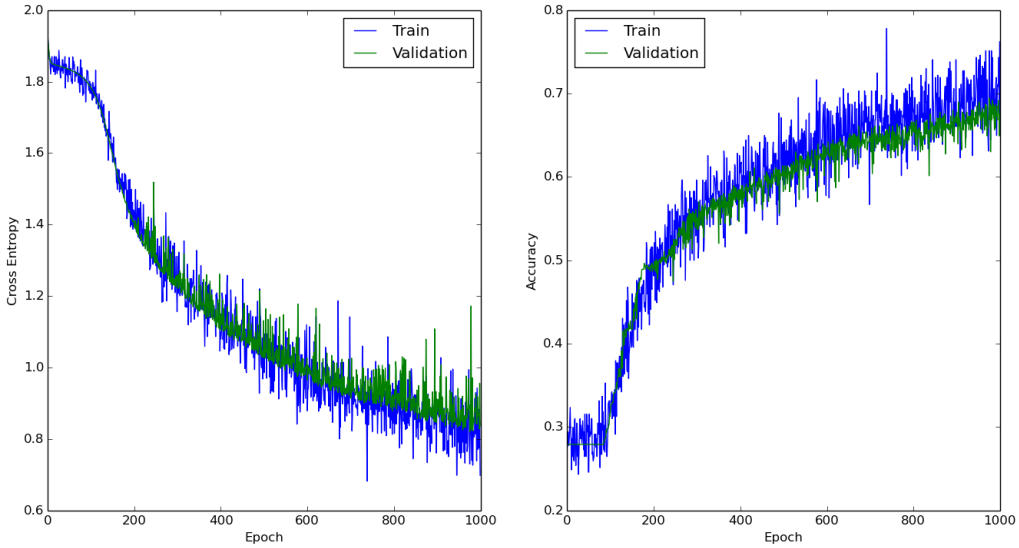

Figure 10: **epsilon=0.01, momentum=0.0, batch size=500**

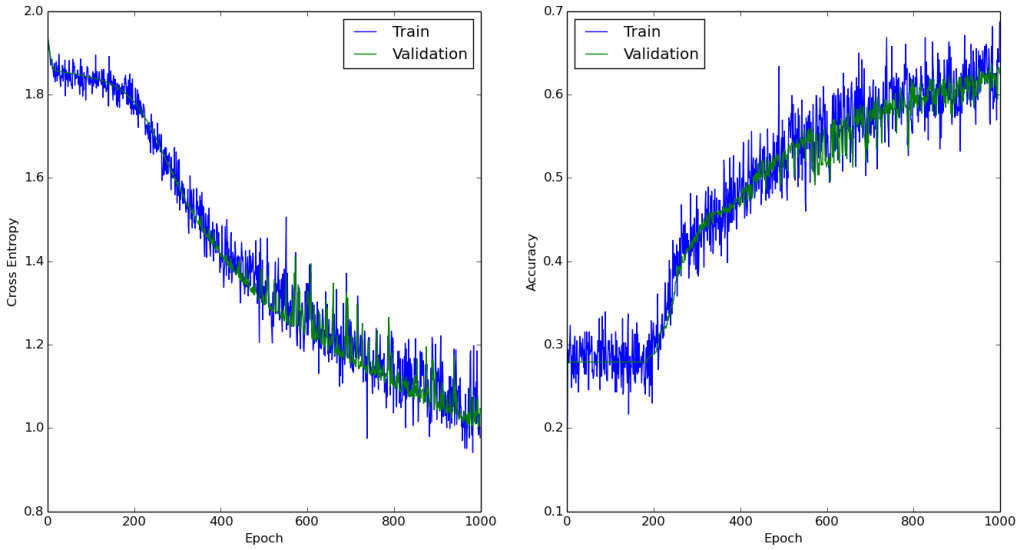

Figure 11: **epsilon=0.01, momentum=0.0, batch size=1000**

What I conclude is that the smaller batch size results in worse accuracy and bad cross-entropies. The higher batch size values result in higher accuracies and lower cross-entropies up to batch size=100. Also, for the case of batch size=1000, accuracy of validation set is very close to that of training set as well their cross entropies. This is because the training accuracies become lower. Also, for batch size values higher than 100, the accuracies converge slower. So the best value for **batch size=100**.

CNN

I ran CNN code for five different epsilon values and then compared their accuracies and their cross-entropies. Below are the results for each run.

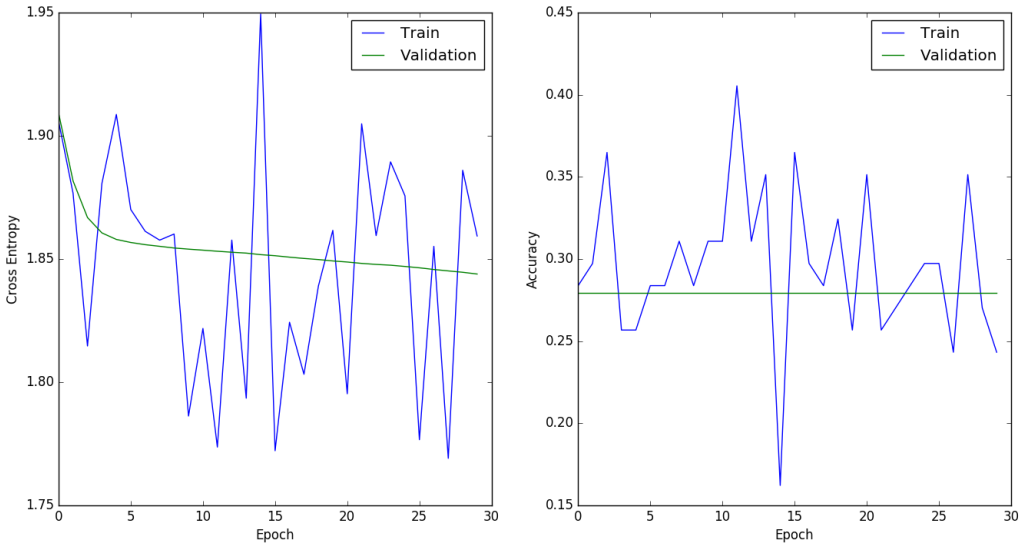

Figure 12: **epsilon=0.001, momentum=0.0, batch size=100**

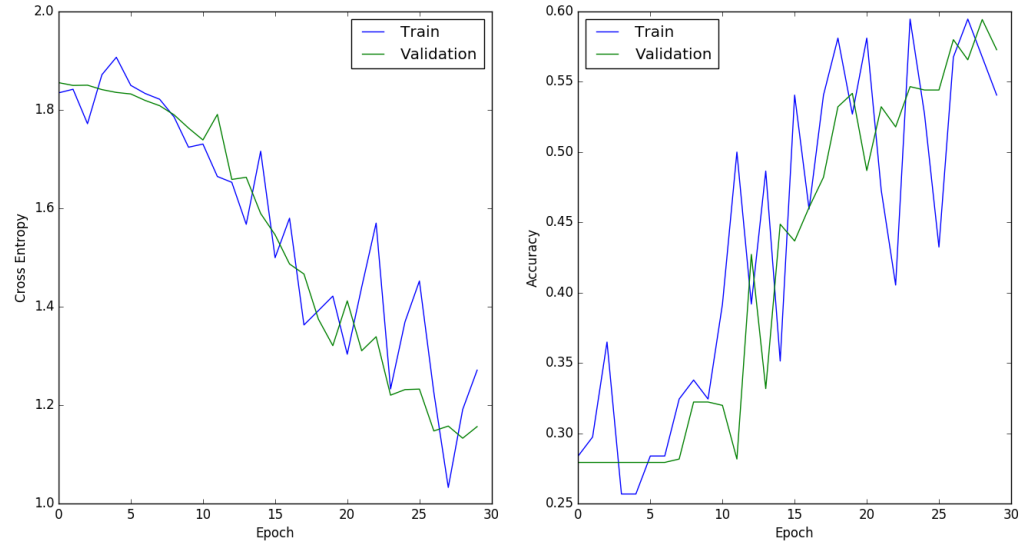

Figure 13: **epsilon=0.01, momentum=0.0, batch size=100**

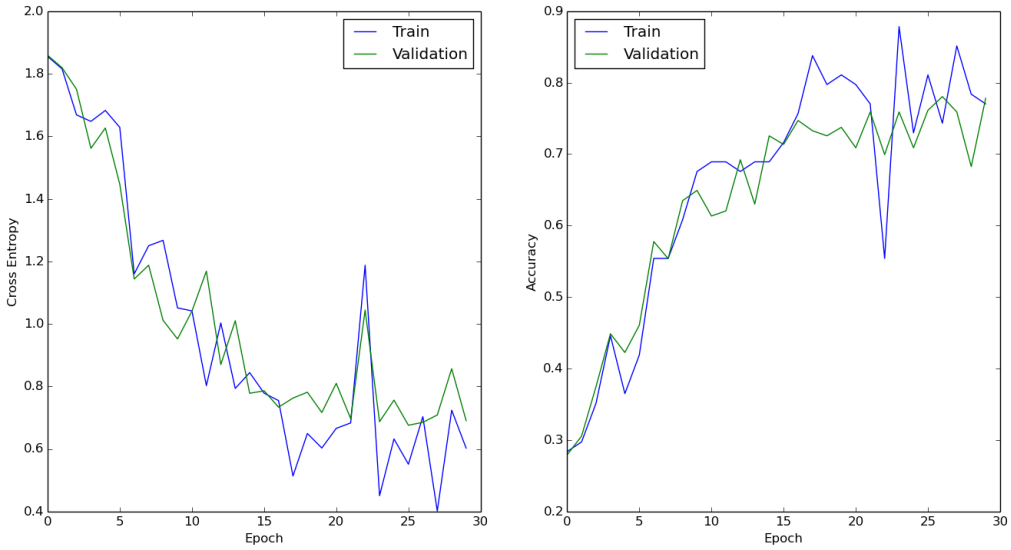

Figure 14: **epsilon=0.1, momentum=0.0, batch size=100**

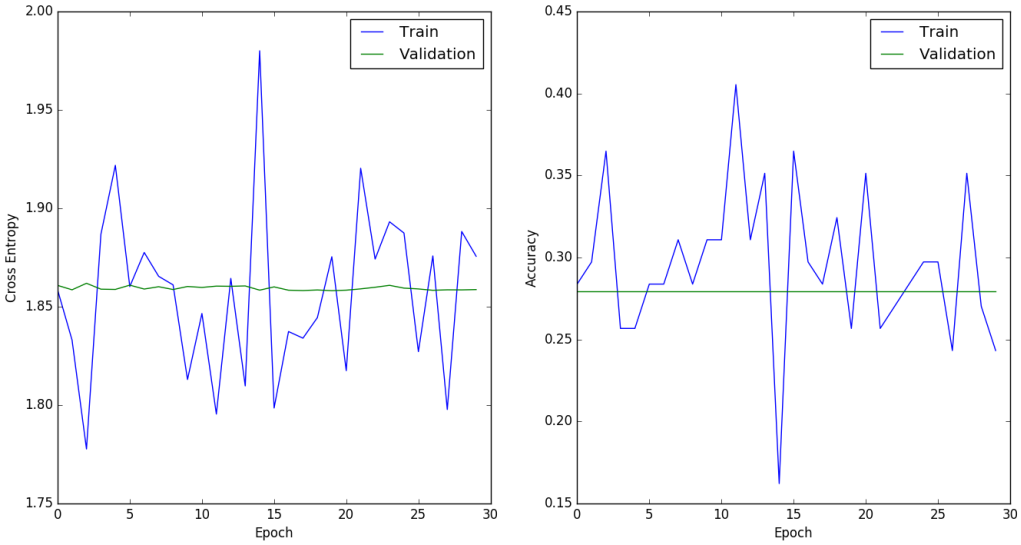

Figure 15: **epsilon=0.5, momentum=0.0, batch size=100**

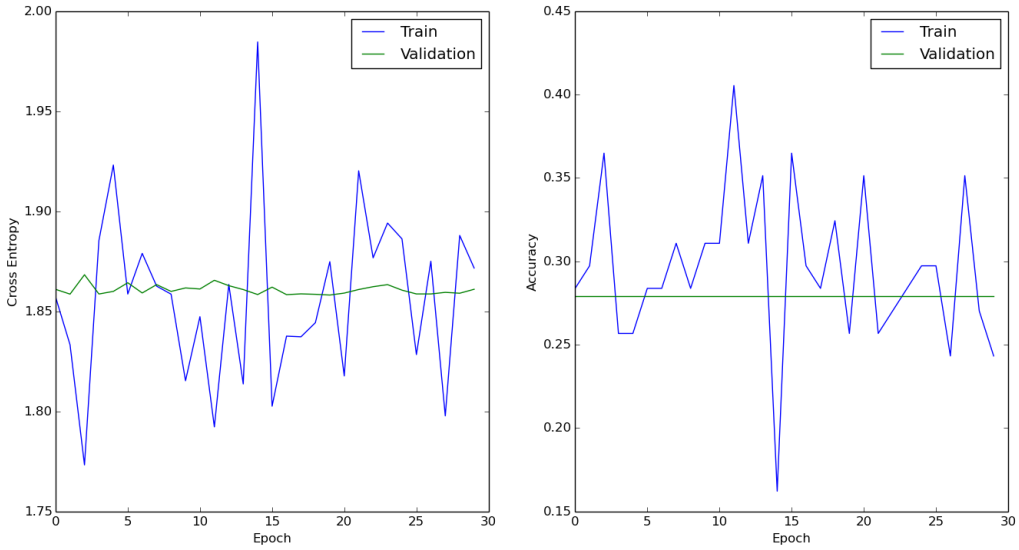

Figure 16: **epsilon=1.0, momentum=0.0, batch size=100**

Based on the figures above, **eps = 0.1** returns the best results using CNN method.

Fixing **eps = 0.1**, I ran CNN code using three different momentum values and the results are shown in figures below:

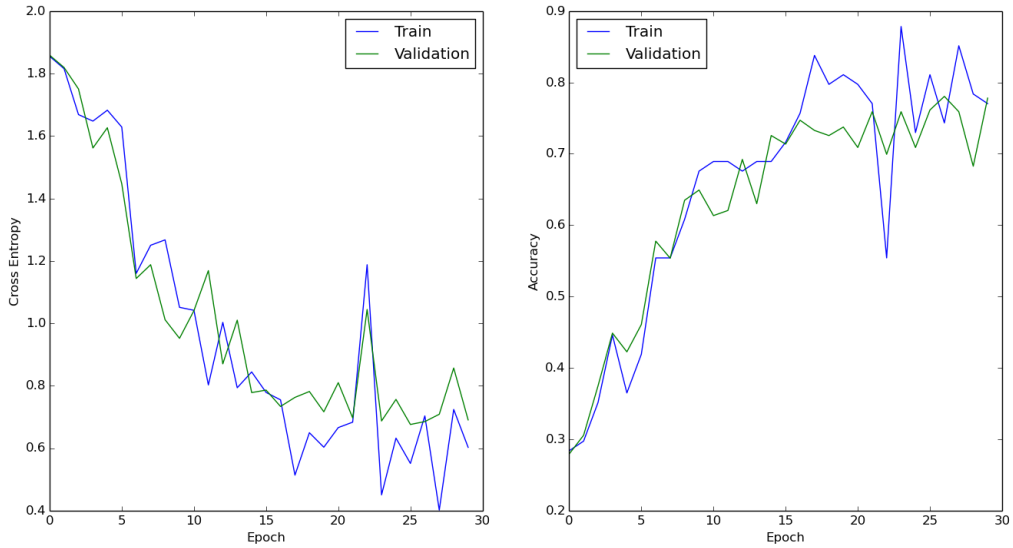

Figure 17: **epsilon=0.1, momentum=0.0, batch size=100**

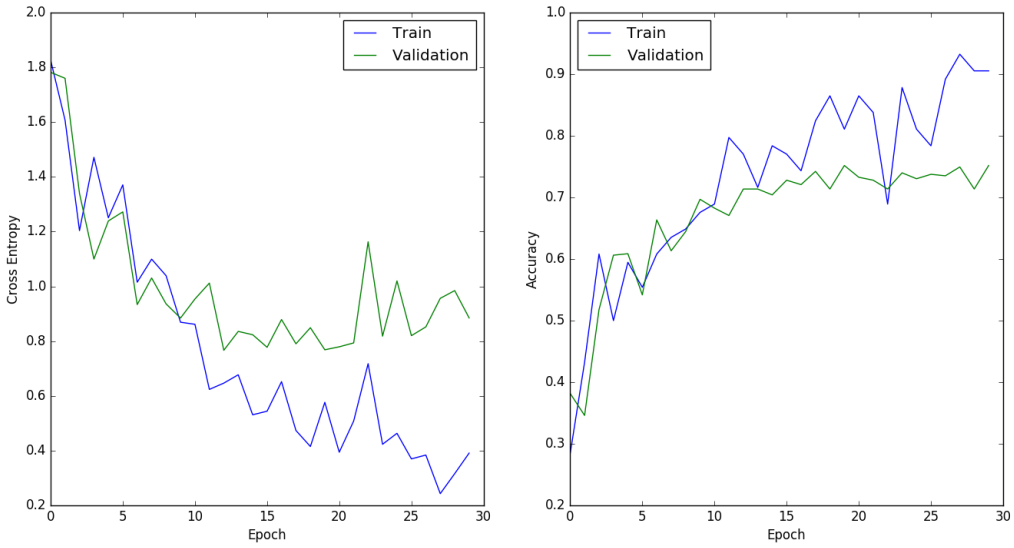

Figure 18: **epsilon=0.1, momentum=0.5, batch size=100**

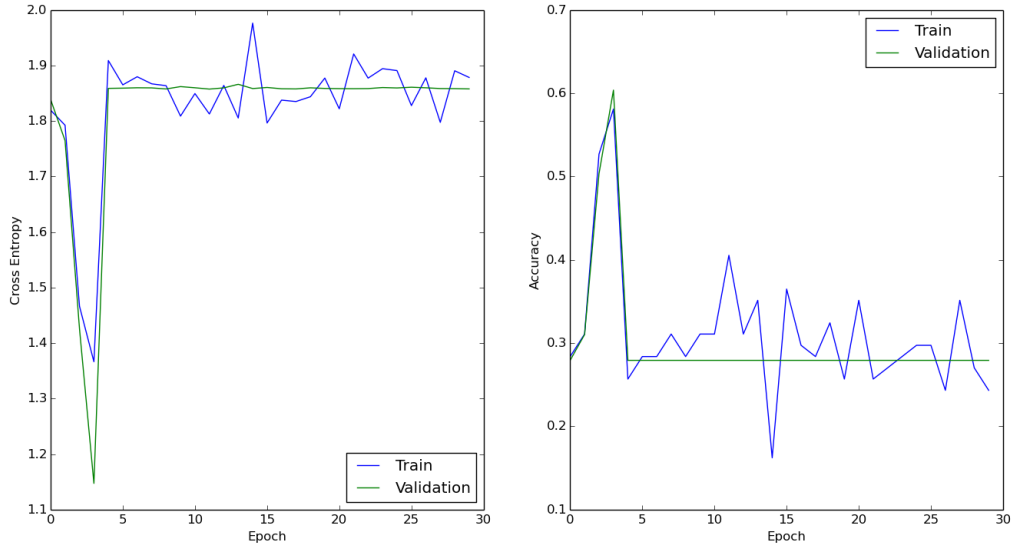

Figure 19: **epsilon=0.1, momentum=0.9, batch size=100**

As we see from the figures above, the best model is a result of using **momentum=0.0**. Using momentum=0.9 results in very bad accuracy and cross-entropy and using momentum=0.5 has lower accuracy and higher CE than when the momentum is 0.

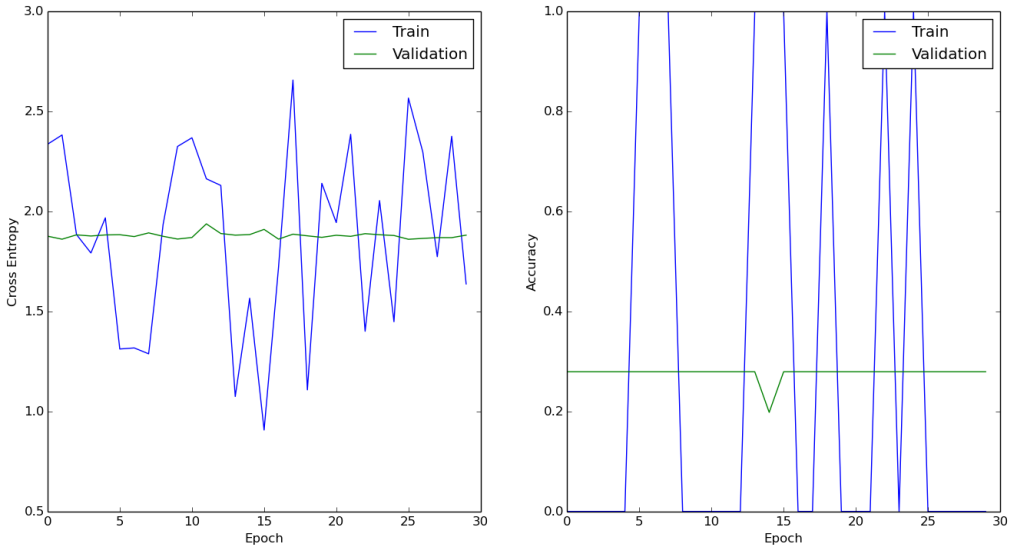

Figure 20: **epsilon=0.1, momentum=0.0, batch size=1**

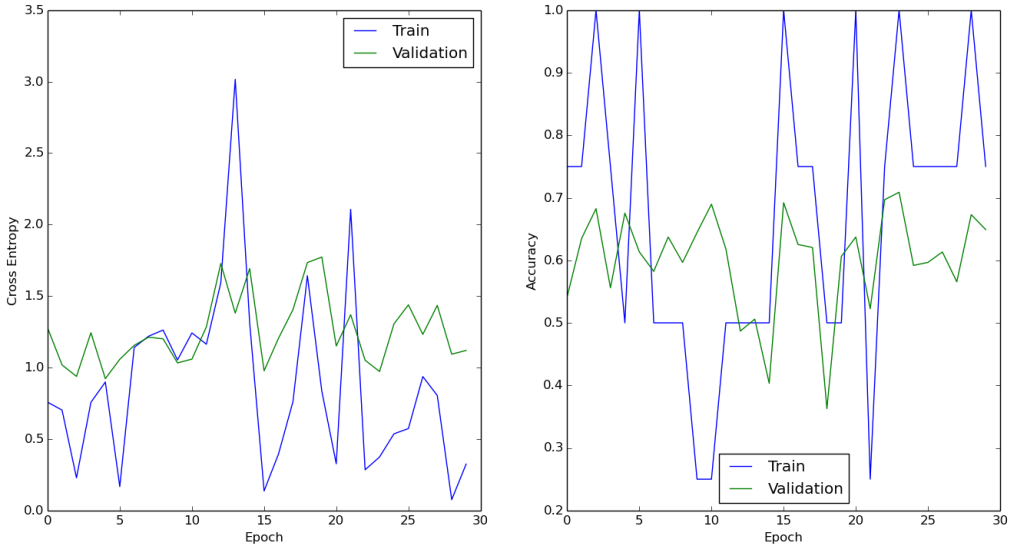

Figure 21: **epsilon=0.1, momentum=0.0, batch size=10**

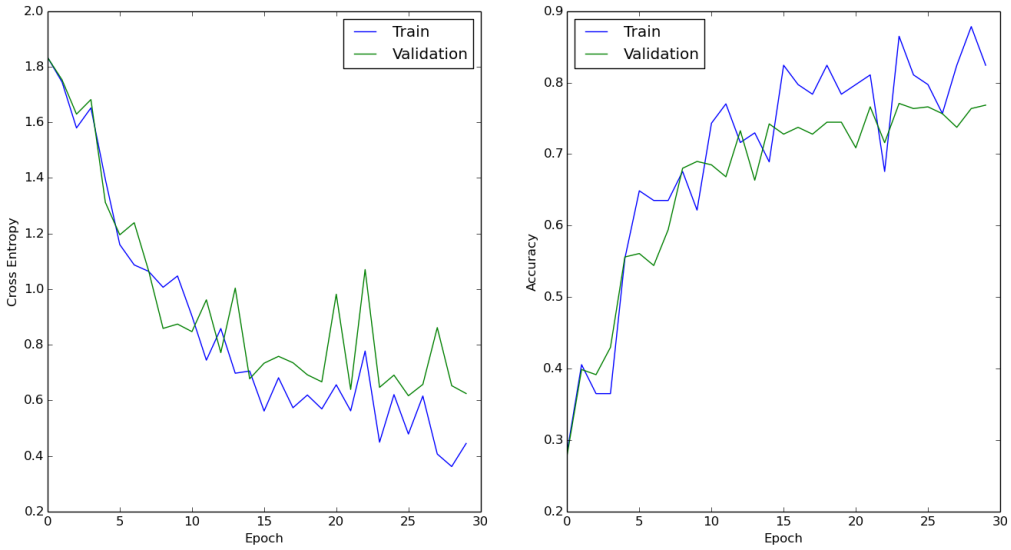

Figure 22: **epsilon=0.1, momentum=0.0, batch size=100**

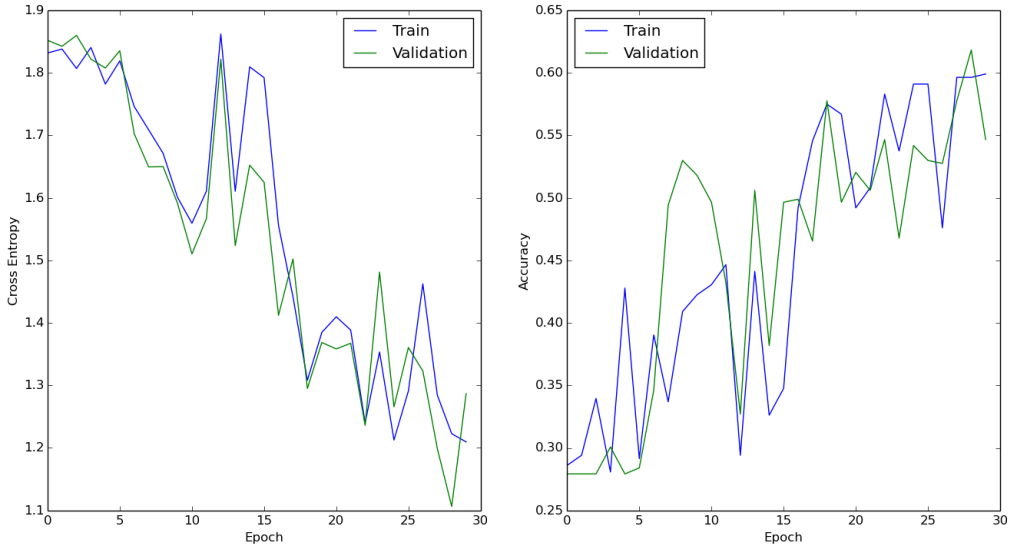

Figure 23: **epsilon=0.1, momentum=0.0, batch size=500**

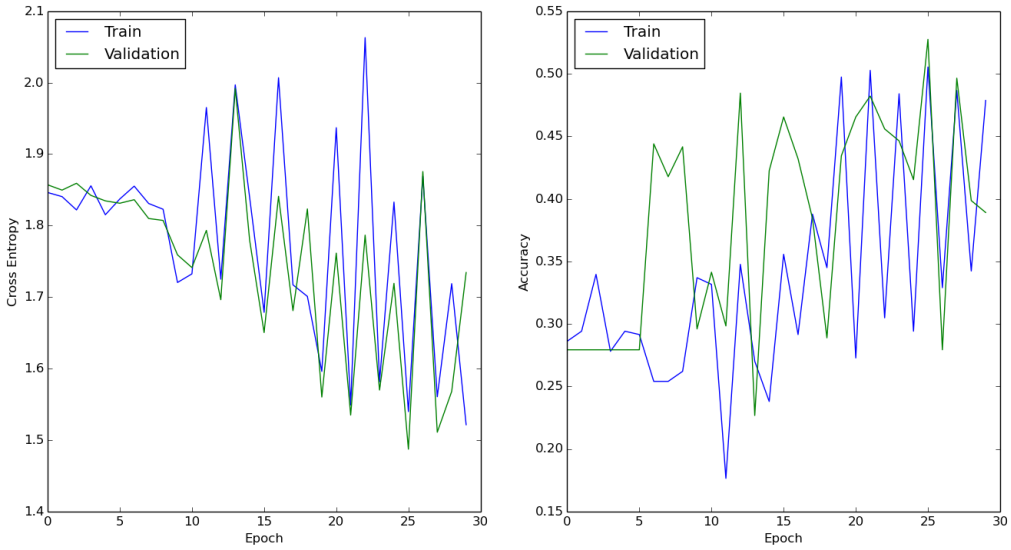

Figure 24: **epsilon=0.1, momentum=0.0, batch size=1000**

Looking at the figures above, we see that batch size=100 has the best results since the accuracy of the validation set is higher and it has lower cross entropy. Batch size=500 and batch size=1000 have lower accuracies around 0.45 -0.6 which is lower than 0.8 for the 100 batch size. Accuracy for batch size = 10 is about 0.6 and stays the same the whole time with no improvement and batch size=1 generates accuracy of as most 0.3 and does not improve either and has high cross-entropy above 1.5.

3.3

NN

Keeping the momentum fixed and equal to 0.9, and having eps = 0.01 for NN, below are the results I got when changing the number of hidden units for each layer. The three combinations I tried are [16,32], [32,32] and [32,16] to see if it matters if the first layer has more units, or if it matters if they are the same or the second layer has more units. Results are shown below:

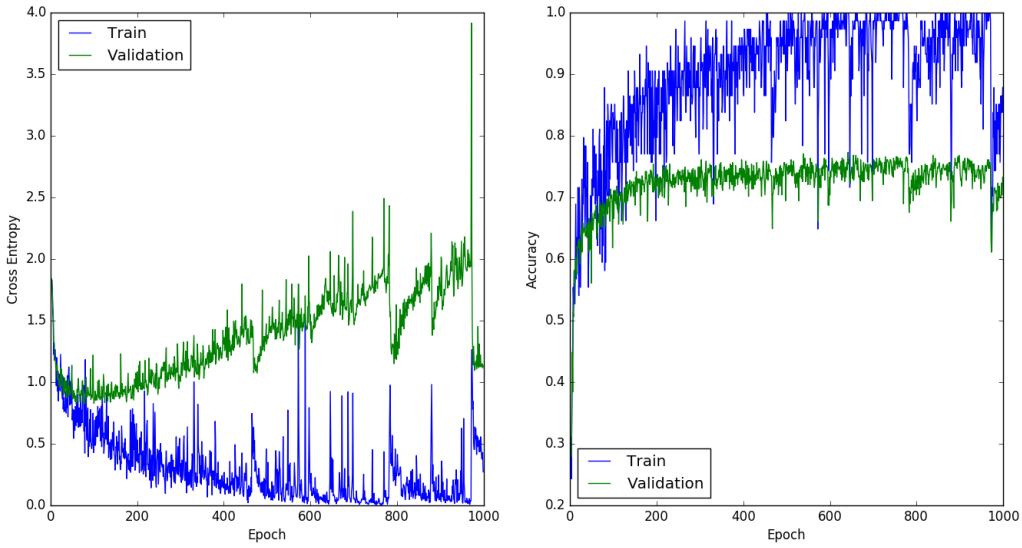

Figure 25: **[16,32],epsilon=0.01, momentum=0.0, batch size=100**

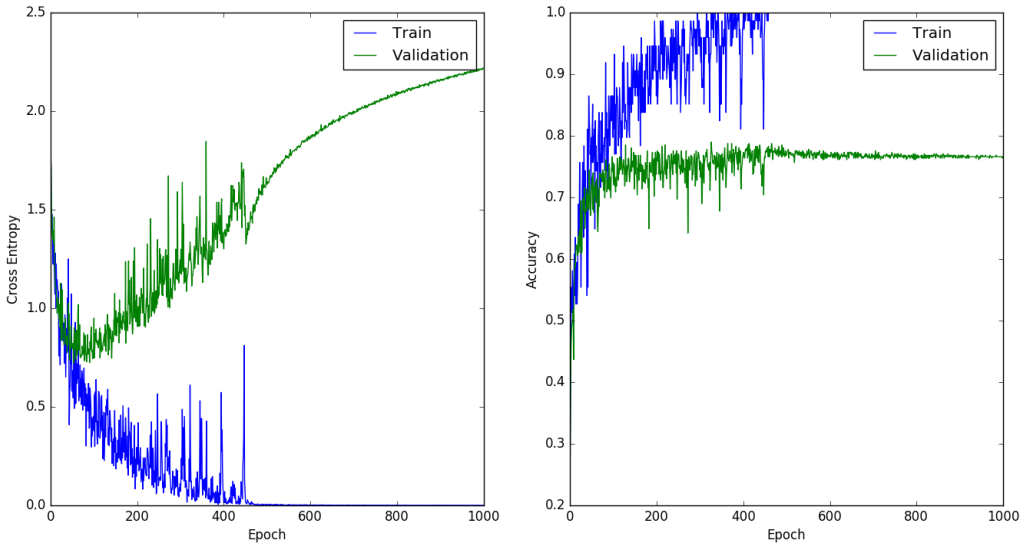

Figure 26: **[32,32],epsilon=0.01, momentum=0.0, batch size=100**

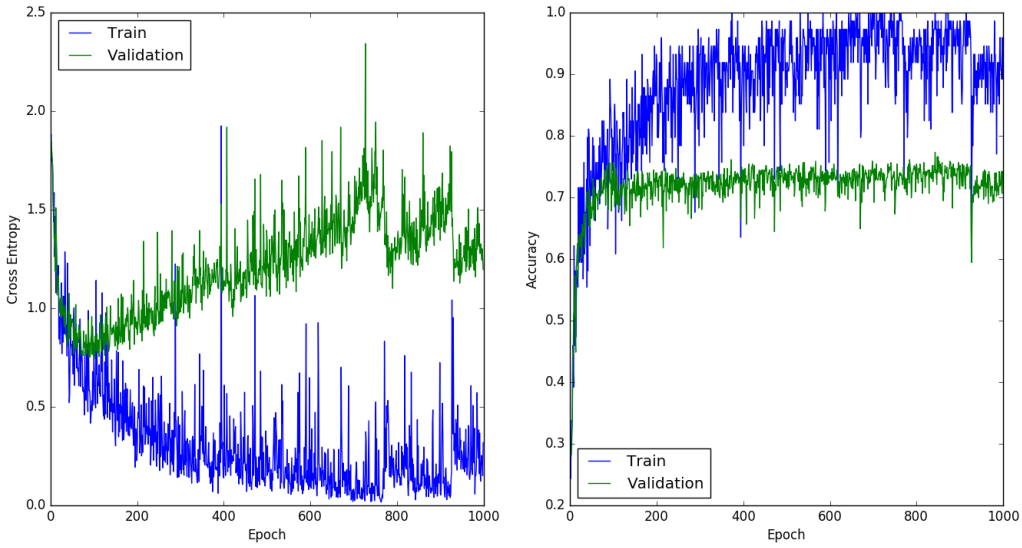

Figure 27: **[32,16],epsilon=0.01, momentum=0.0, batch size=100**

As we see above, the accuracy of the validation set is more or less the same when using different number of units in the hidden layers and the small variation could be due to the randomness of the weight initialization. Cross-entropy of validation set starts increasing when there is the same number of units in both

CNN

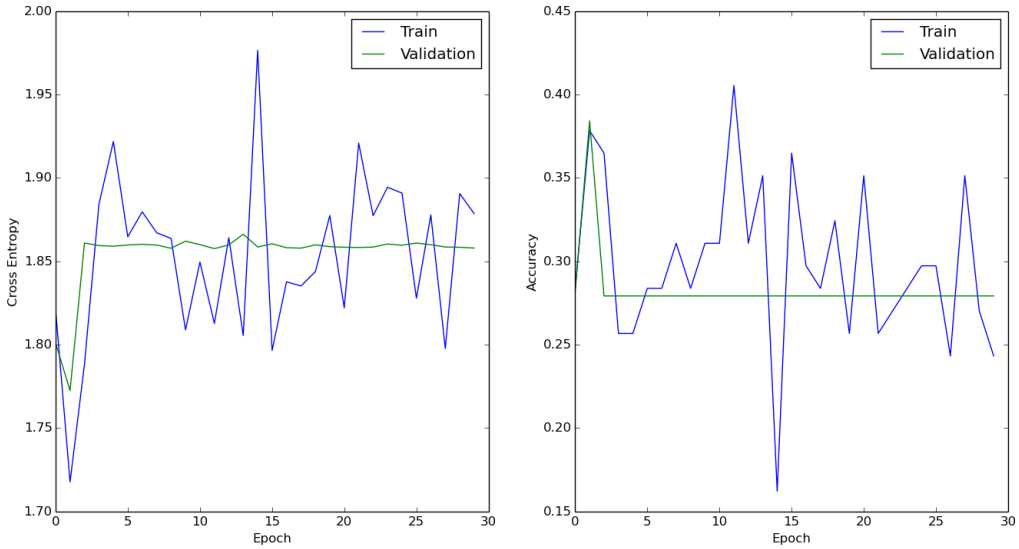

Figure 28: **[8,16],epsilon=0.1, momentum=0.9, batch size=100**

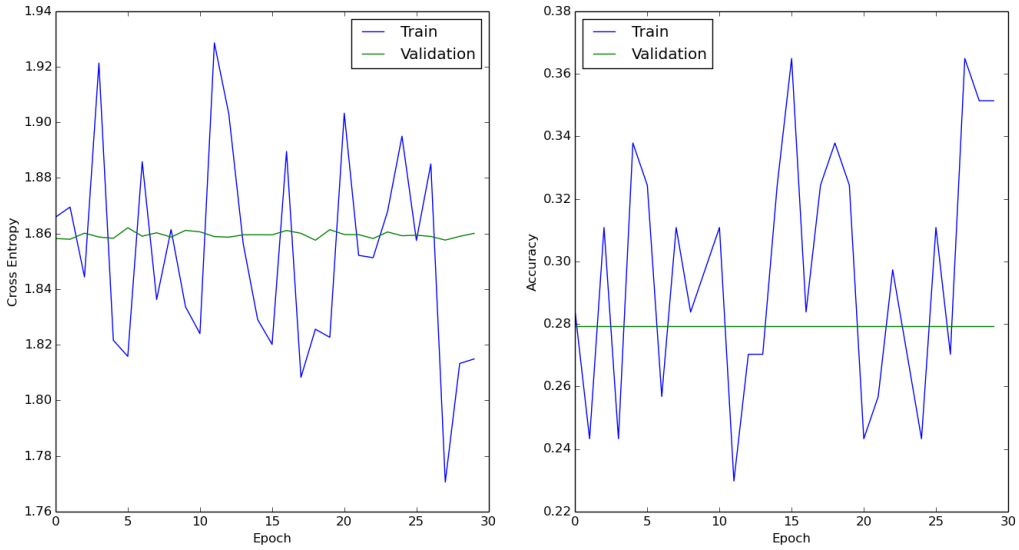

Figure 29: **[16,16],epsilon=0.1, momentum=0.9, batch size=100**

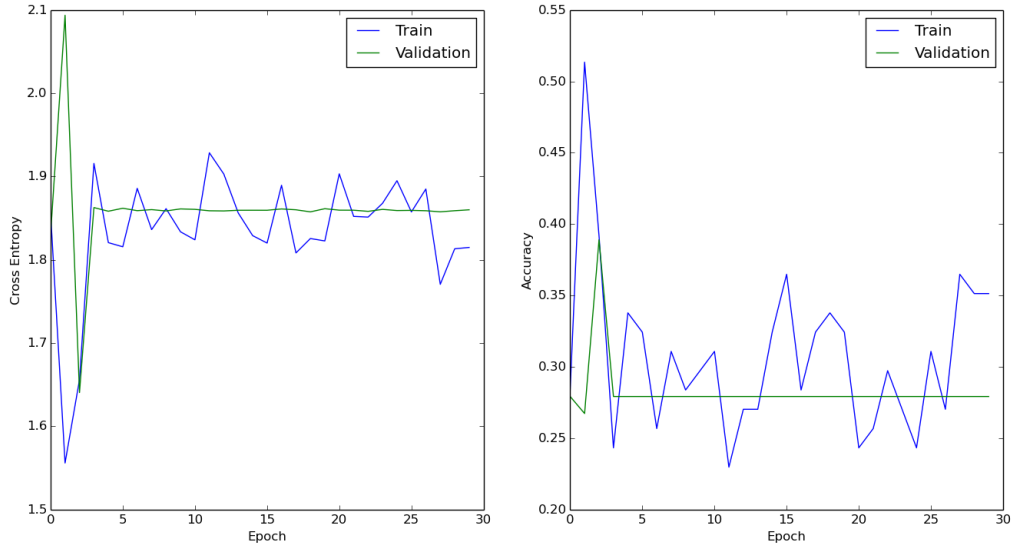

Figure 30: **[16,8],epsilon=0.1, momentum=0.9, batch size=100**

We see that learning rate of eps = 0.1 is too high when using momentum=0.9 in this case which caused the model to miss the optimum value and not learn and not improve.This means that we need to change eps to be smaller so I chose **eps=0.1, momentum=0.9** and then tried these filters again. The new results are shown below:

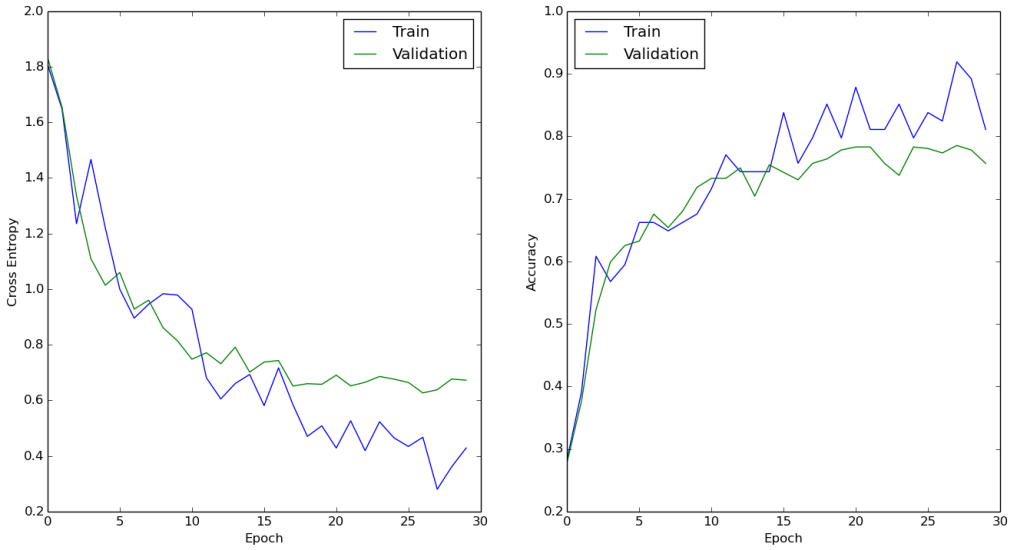

Figure 31: **[8,16],epsilon=0.01, momentum=0.9, batch size=100**

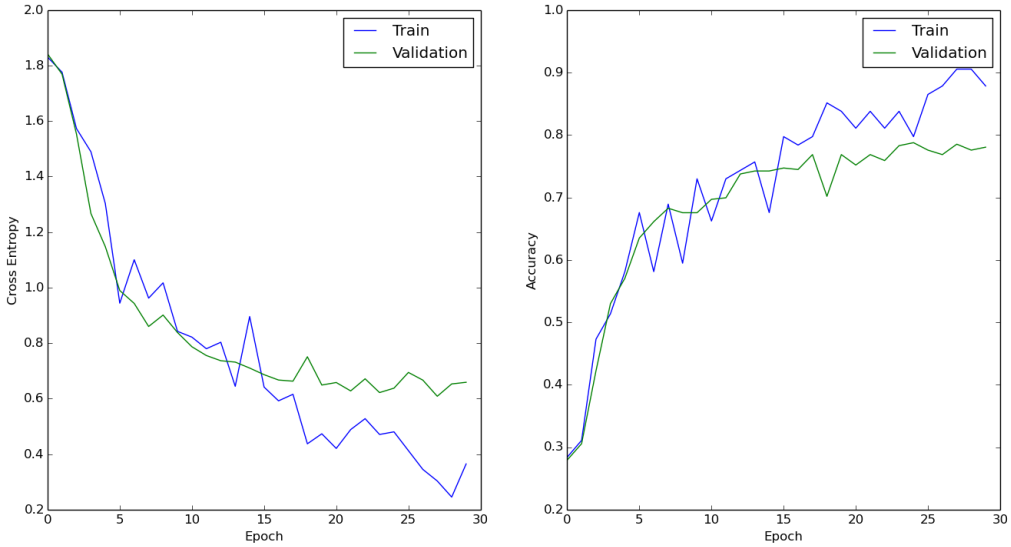

Figure 32: **[16,16],epsilon=0.01, momentum=0.9, batch size=100**

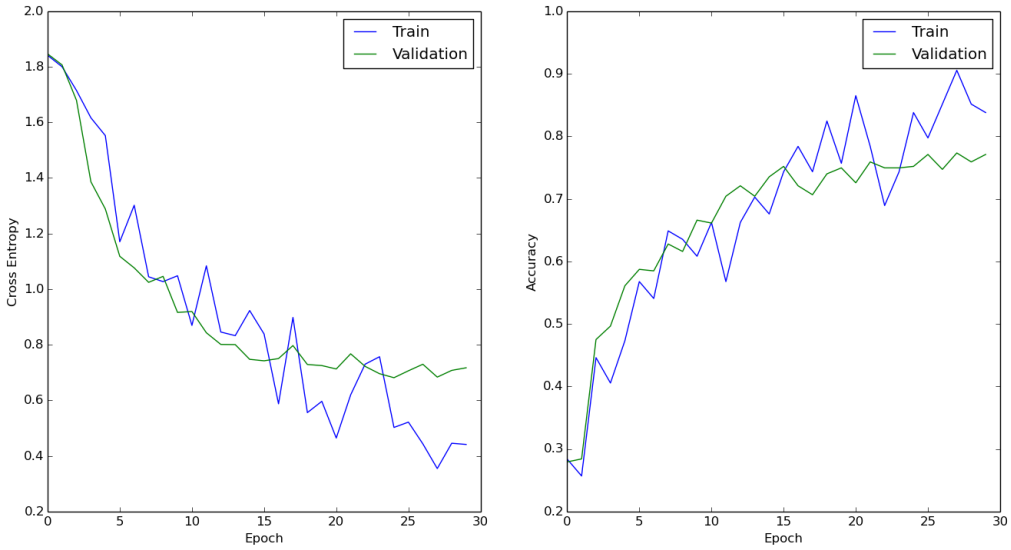

Figure 33: **[16,8],epsilon=0.01, momentum=0.9, batch size=100**

We see that we get much better results when I adjusted the learning rate. Also, as we increase the number of hidden units, the code runs slower because it has to do more calculations. Larger number of hidden units result in better values but too large of value results in over-fitting. Too small of number of hidden units results in not capturing all of the features as well. So we need to decide on the number of hidden units such that it does not over-fit or does not miss capturing features.

3.4

The number of parameters we need to calculate is the obtained in the following way:
size(W1) + size(W2) + size(W3) + size(b1) + size(b2) + size(b3). So I printed the size of these arrays in the initialization function in both NN and CNN.
For NN, the size of each weight and bias is as follows:
size(W1) = $2304 \times N_1$
size(W2) = $N_1 \times N_2$
size(W3) = $N_2 \times 7$
size(b1) = N_1
size(b2) = N_2
size(b3) = 7

For CNN case, the weights and bias sizes are given as follows:
size(W1) = $5 \times 5 \times 1 \times N_1$
size(W2) = $5 \times 5 \times N_1 \times N_2$
size(W3) = $64 \times N_2$
size(b1) = N_1
size(b2) = N_2

size(b3) = 7

So I wrote a function to sum these up and then tried a few different values of N_1 and N_2 for can and then tried to get a close value for nn with trial and error. The code is shown below:

```
def func_cnn(N1, N2):
return 26*N1 + 449*N2 + 25*N1*N2 + 7

def func_nn(N1, N2):
return 2305*N1 + 8*N2 + 7 +N1*N2
```

The values I got for nn code are $[N_1, N_2]$ = [7,13] which results in 16337 parameters (including bias) and for can I choose $[N_1, N2]$ = [10,23] which results in 16344 parameters (including bias). The number of parameters are very similar.

The sum of all of these would be the number of parameters in each method, so we need to choose N_1 and N_2 values for each method such that both methods have similar number f parameters so that we can compare the results better.

Using these values for the number of hidden values and filters, I got the results shown below:

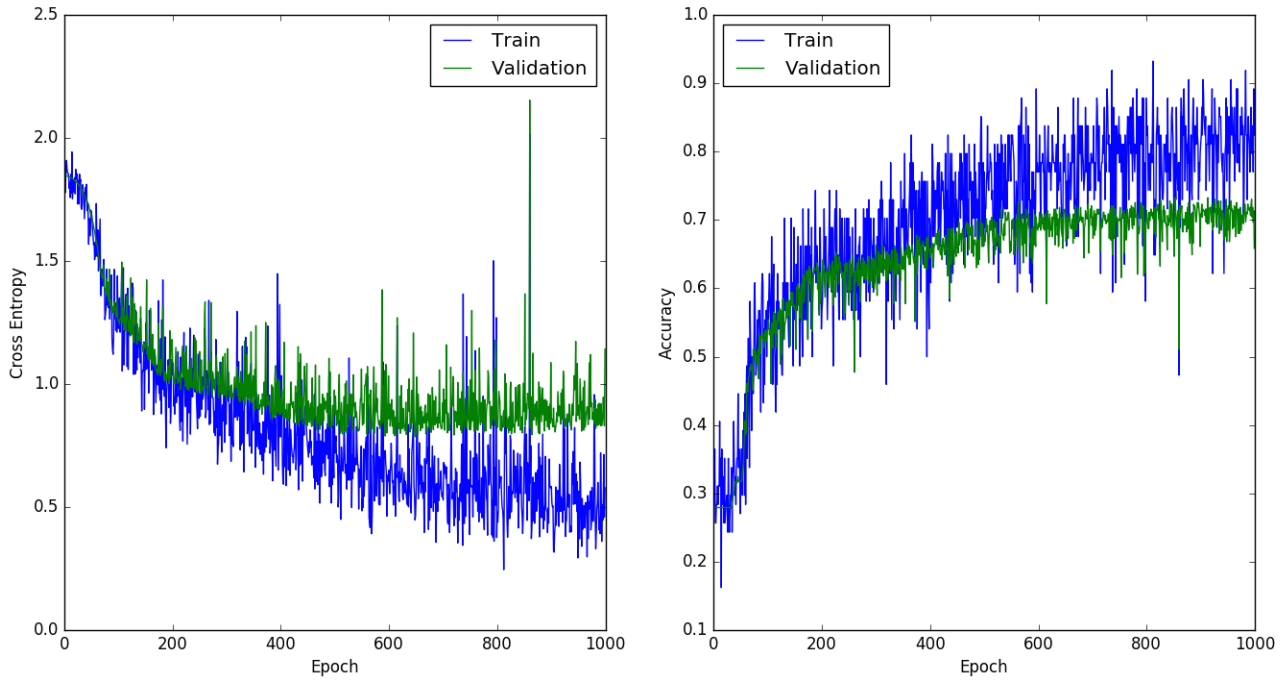

Figure 34: Cross-entropy and accuracy using nn method using hidden layers = [7,13].

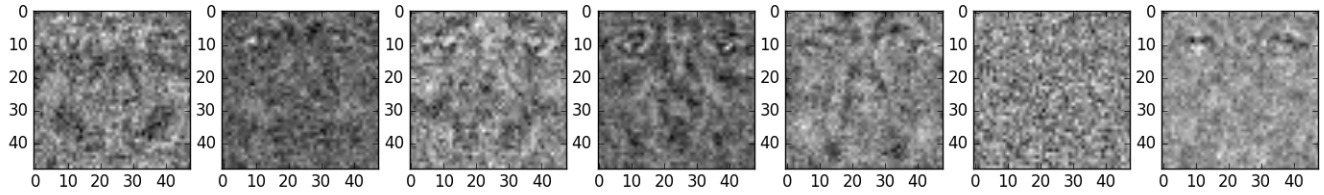

Figure 35: Weights of the first layer using nn method using hidden layers = [7,13].

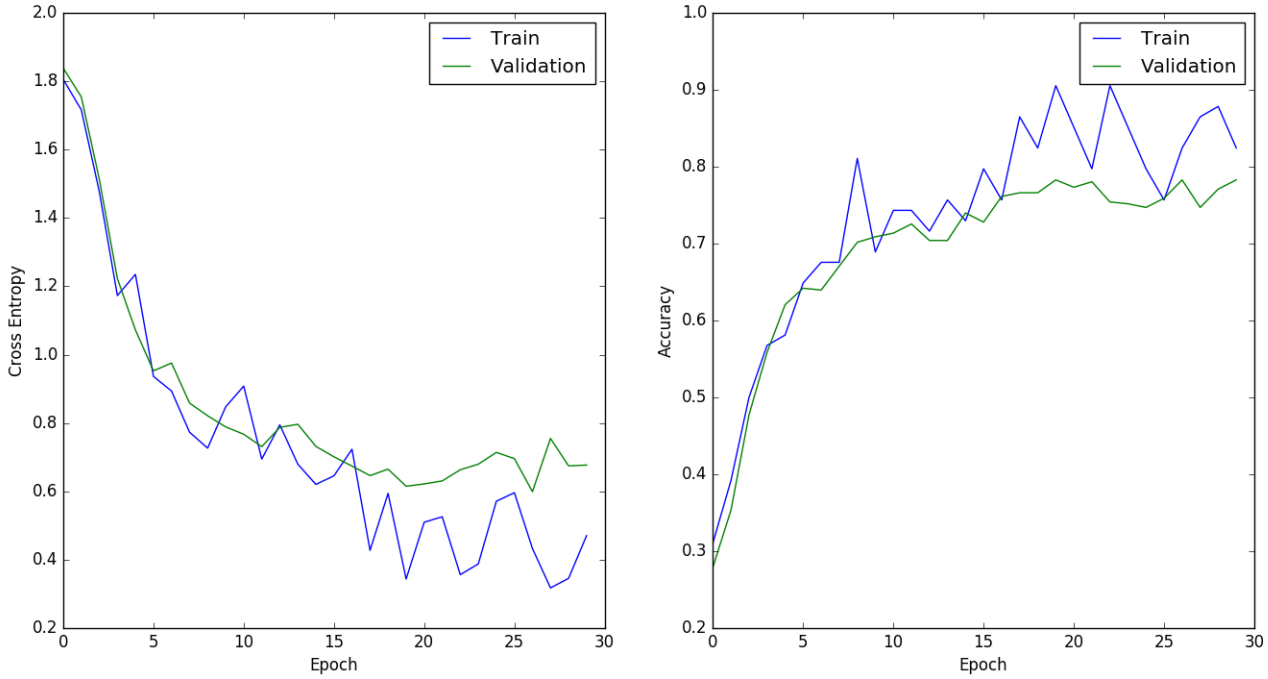

Figure 36: Cross-entropy and accuracy using nn method using hidden layers = [10,23].

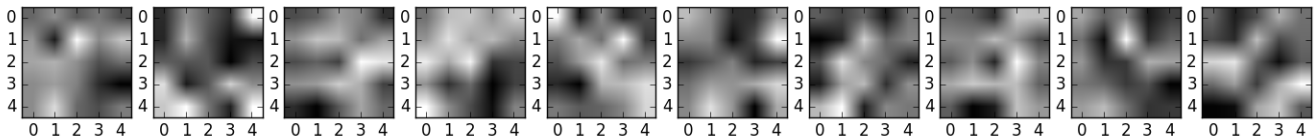

Figure 37: Weights of the first layer using cnn method using hidden layers = [10,23].

The first layer we are looking at for the nn method, we can see human face fewtures in the images. We can see nose and eyes in some of them which means that in nn method it tries to find parts of the whole

image in this case looks for eyes or nose. However, in cnn method, we can see that we cannot distinguish any face features and we can only see vertical and horizontal lines which means that this layer is looking for vertical and horizontal lines only. This is the difference between the two methods.

CNN method is better than NN in that it is more generalizable. This is because NN method can distinguish particular and specific features such eyes and nose. So in my opinion if we tilt the image or if the images are not all aligned it would be harder for NN to distinguish between images and to classify them. However, in CNN method, we have many different filters in which each filter looks for very generalizable features such as having vertical lines or horizontal lines or colour, which makes distinguishing of images better than NN method.

3.5

Below are three images of bad images where the maximum prediction of the image is less than 0.55:

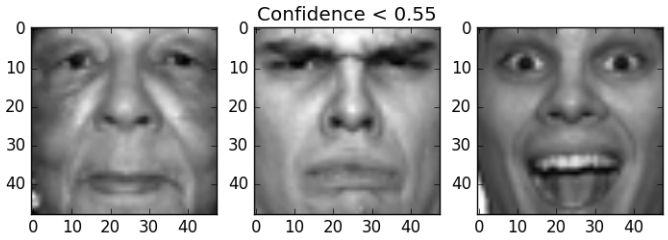

Figure 38: Bad predictions where highest prediction of each image 0.55.

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.

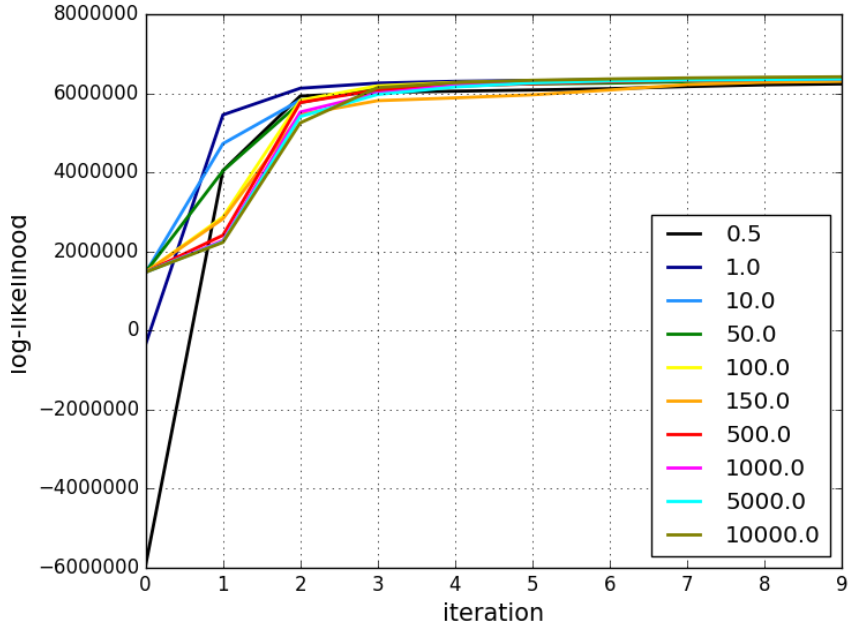

Figure 39: Log-likelihood as a function of number of iterations for different values of randConst parameter.

As we see in Figure 39, 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:

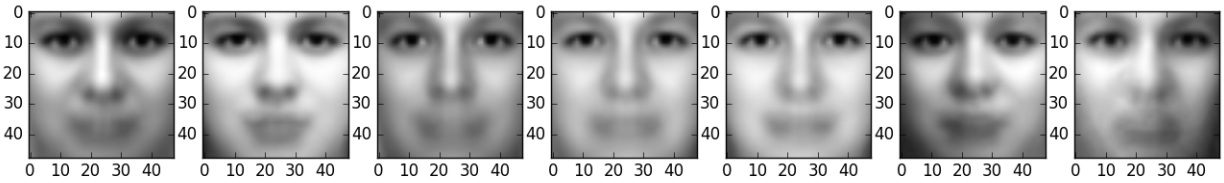

Figure 40: Mean of the images- Is is blurry because it is average.

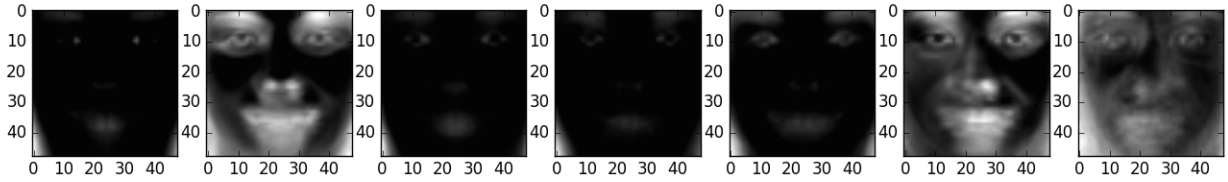

Figure 41: Variance of the images-black means lower variance and white means higher variance.

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 42, the values of Gaussian mixture coefficients are given vs. cluster number.

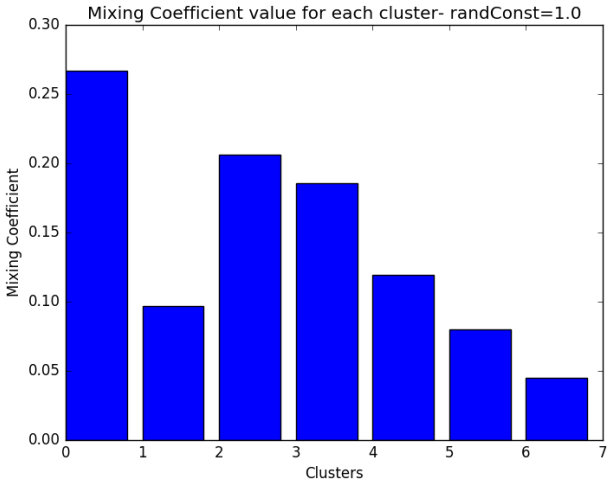

Figure 42: Value of mixture of coefficients for each cluster.

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 45 below.

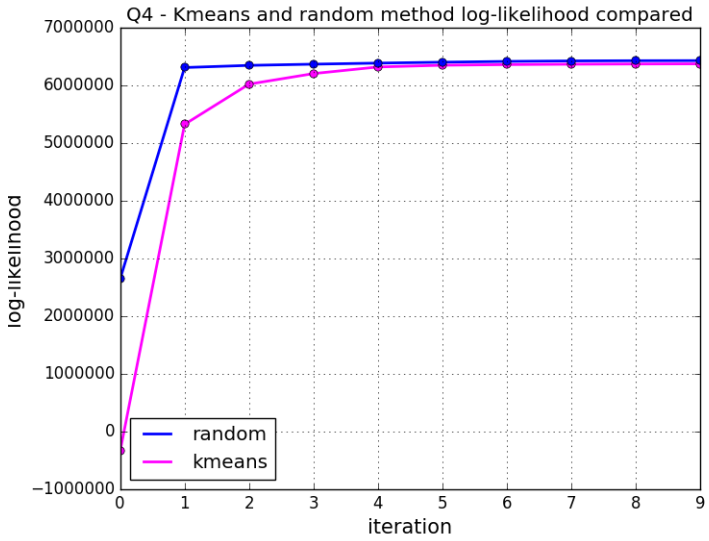

Figure 43: Comparison between log-likelihood using two different methods: Kmeans shown in magenta and randomized method (using randConst) shown in blue. We can see that using means method the log-likelihood converges earlier to a similar value compared to the randomized method.

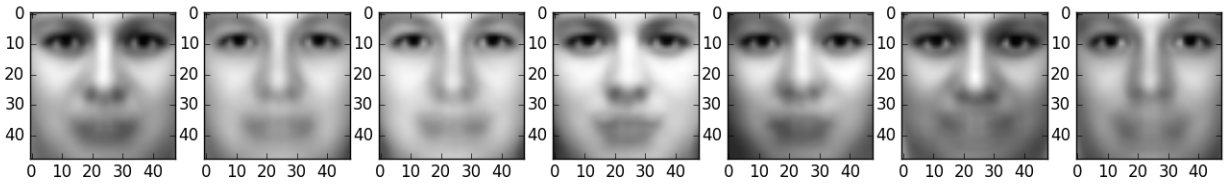

Figure 44:

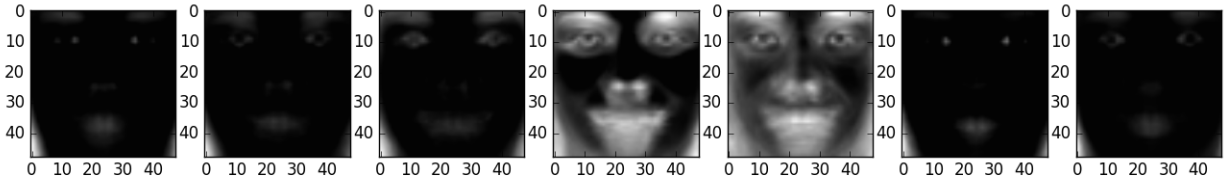

Figure 45:

The speed of means method is faster than the random method and this is due to .

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 **??** shows the for all three cases of training set, validation set and test set.

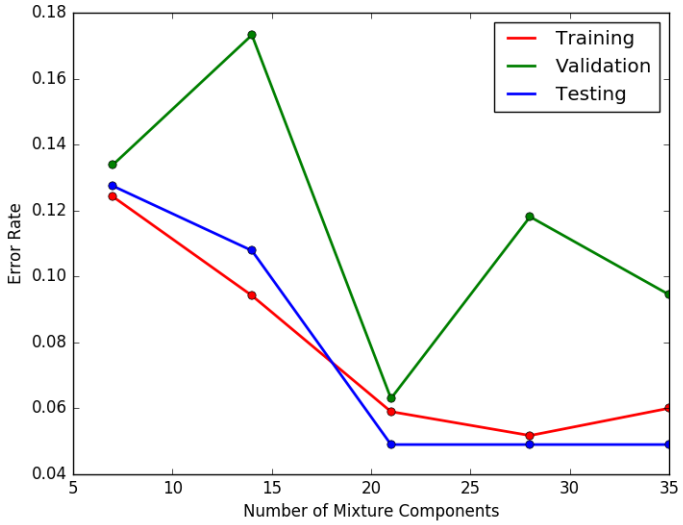

Figure 46:

Answers to the questions:

(b) We find that the error rates on the training set generally decreases as the number of clusters increases. This is because it is easier to fit to larger number of Gaussians. It obviously over-fits the training data if the number of cluster is very large but it still fits the training set well so the error rate decreases.

(c) The error rate for test set decreases as we increase the number of clusters. This is not the case for validation set as we see (the error rate increases for higher cluster numbers). This is because we are over-fitting the training set by having larger number of clusters and therefore the model does not work well on the validation set for larger number of clusters and the reason it works for the test set is because of the input we have for test set. I think we are just lucky that it works well on the test set and usually this is not the case.