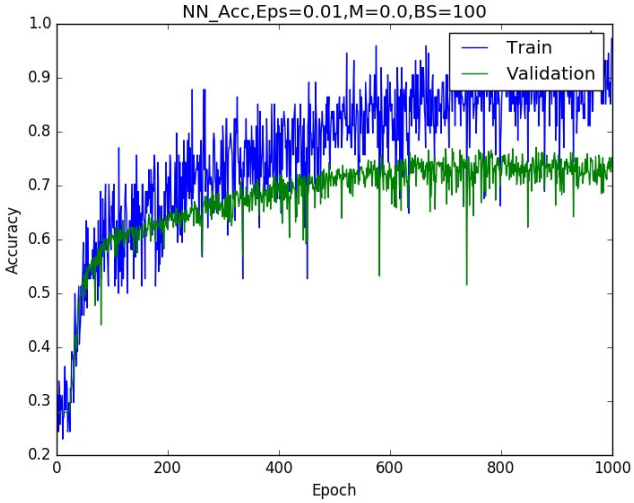
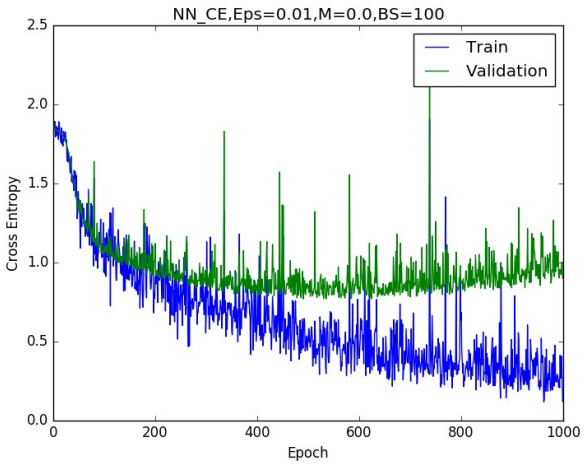
Joel Cheverie
1002924393
CSC 2515: Assignment 2

3.1

Default Neural Network

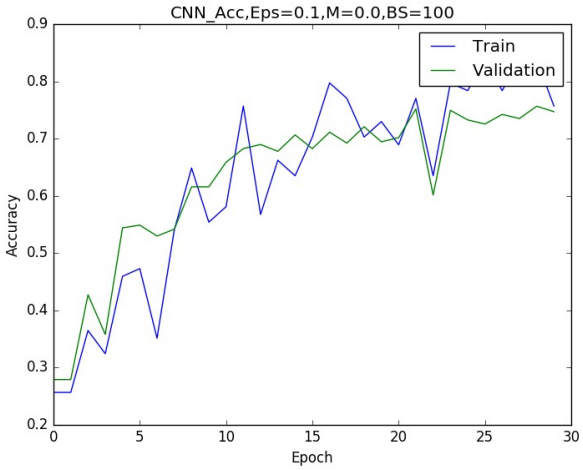


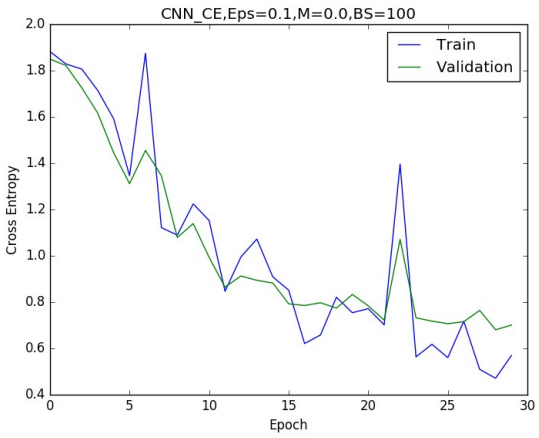


CE: Train 0.37885 Validation 1.06263 Test 0.97500
Acc: Train 0.85507 Validation 0.73031 Test 0.71688

With the default set of hyperparameters, the network is both more accurate and has less loss on the training set versus the validation set. As the cross entropy for validation appears to be increasing well before 1000 epochs, this also implies that the model is likely overfitting the training data. It can likely be stopped well before 1000 iterations, as it appears to converge around epoch 300 or so.

Default Convolutional Neural Network





CE: Train 0.54408 Validation 0.70088 Test 0.67350
Acc: Train 0.81031 Validation 0.74702 Test 0.77662

With the default set of hyperparameters, the network is both more accurate and has less loss on the training set versus the validation set. Surprisingly, the performance is initially better in terms of both cross entropy and accuray on the validation set than on the training set, only flipping around epoch 15. Here it appears that the overall network performance converges around epoch 25, as the validation accuracy and cross entropy remains fairly static from that point onwards. Interestingly, this network is both more accurate and has less loss than the more basic neural network on the validation and test sets, implying that it overfits less and generalizes better.

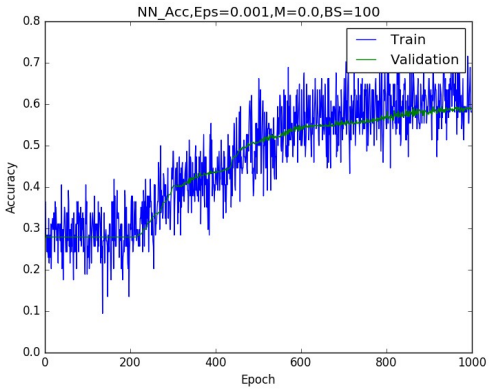
3.2

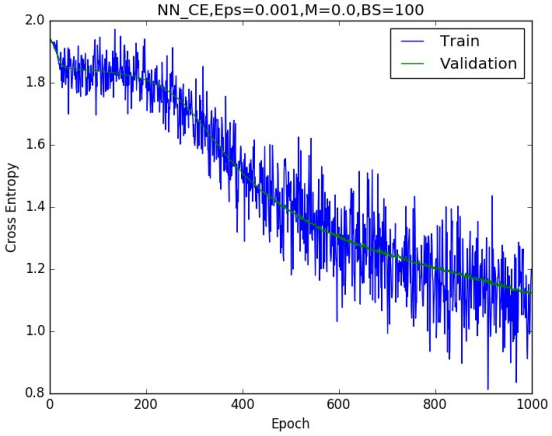
Neural Network

For my experiments with hyperparameters, I chose epsilon values of (0.001, 0.05, 0.1, 0.5, 1.0), momentum values of (0, 0.5, 0.9) and batch sizes of (1, 10, 50, 500, 1000). While varying one hyperparameter, I kept the others at their default values. Here are the relevant plots:

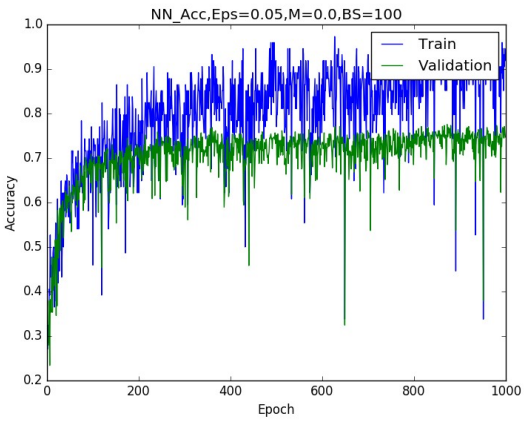
Plots for Epsilon

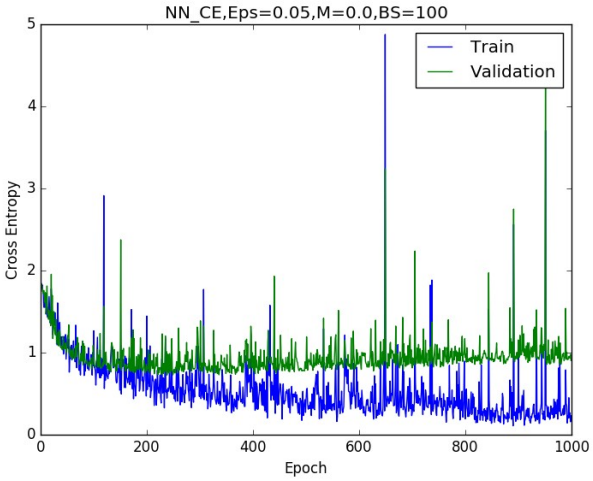
Epsilon = 0.001



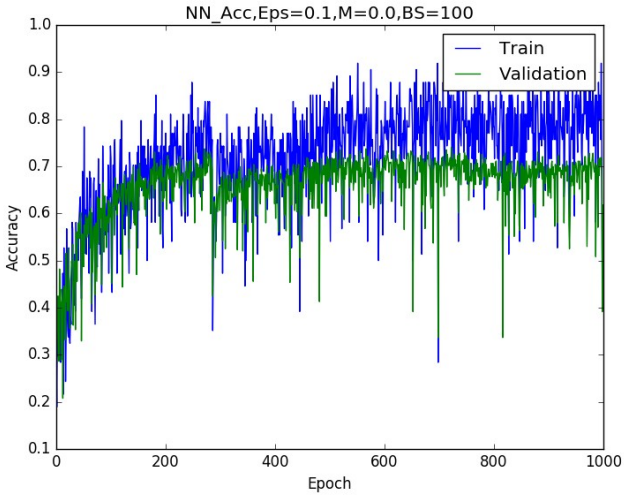


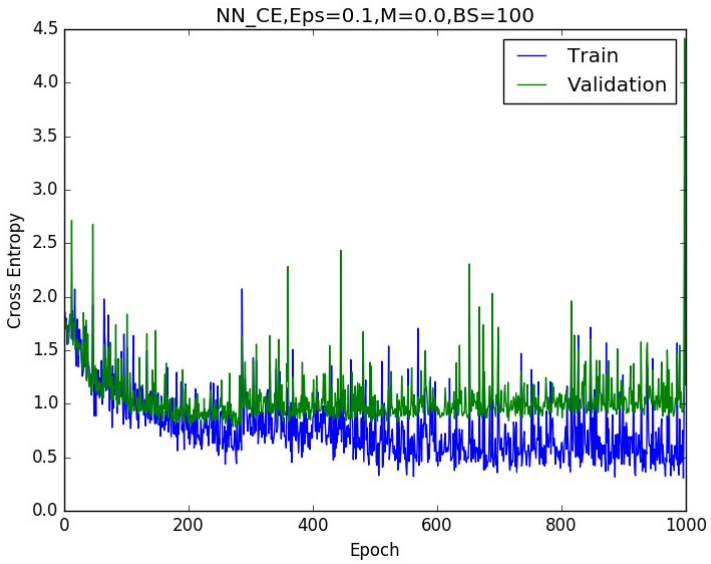
Epsilon = 0.05



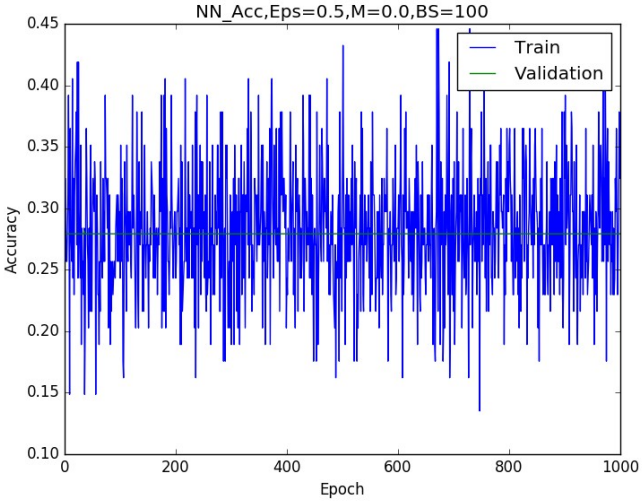


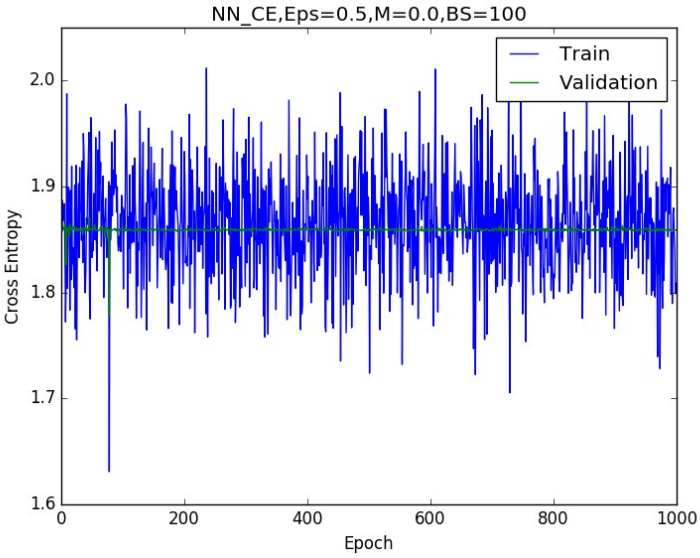
Epsilon = 0.1



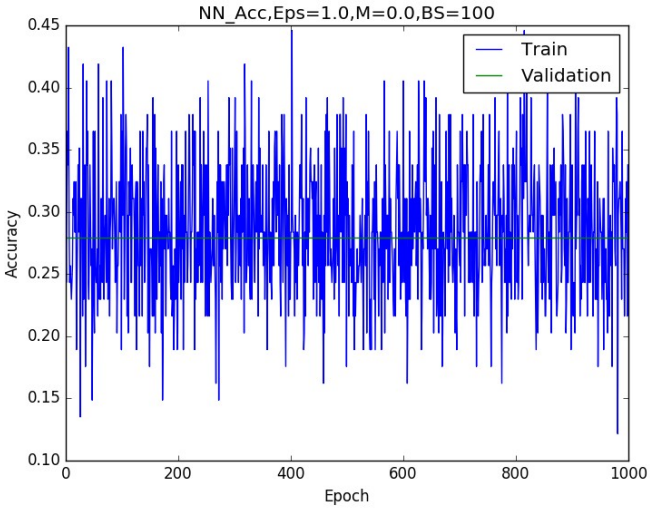


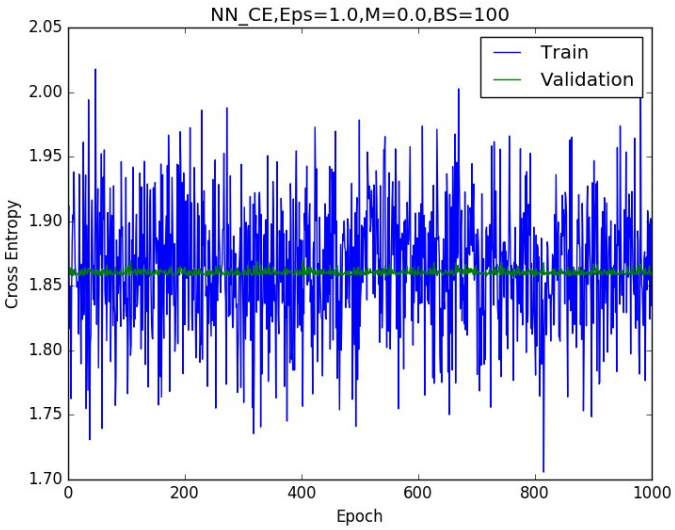
Epsilon = 0.5





Epsilon = 1.0

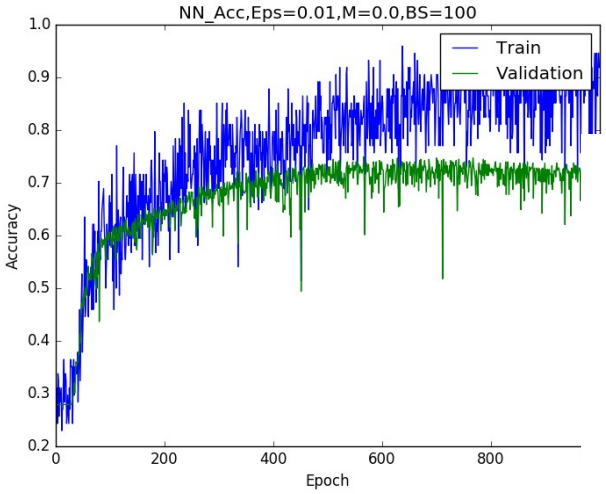


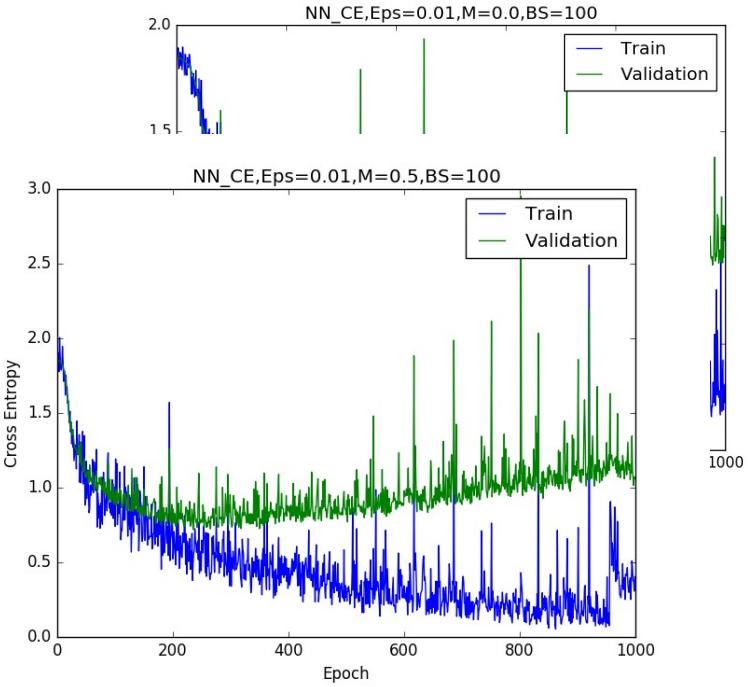


It would appear that as epsilon is increased, the network converges faster. With very high values of epsilon, the network performance erodes however. At 0.5 and 1.0 it is very noisy and does not appear to be learning at all. The performance does not improve at all in these cases. For the epsilon value of 0.001, the network does not seem to converge for accuracy or cross entropy. For the other two values (0.1 and 0.05), it converges faster than the default epsilon value, but seems to perform similarly to the default network (in both accuracy and cross entropy) on the validation set. Hence the main gain here seems to be faster convergence in epoch time by increasing epsilon up to a certain point, but not so much as to prevent learning.

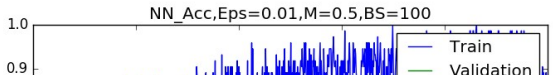
Plots for Momentum

Momentum = 0.0

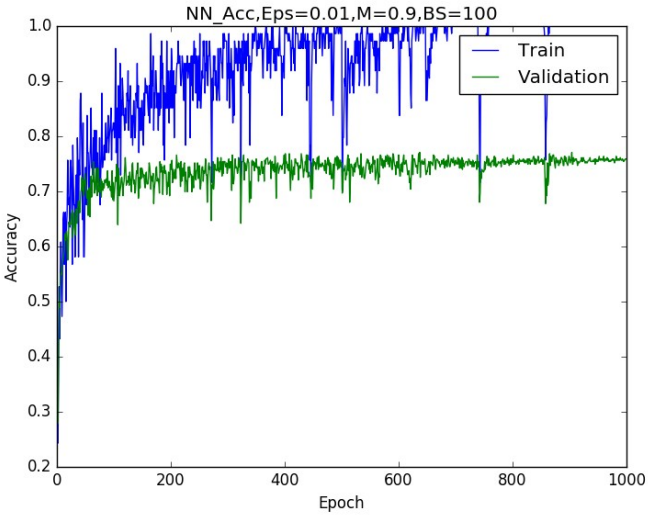


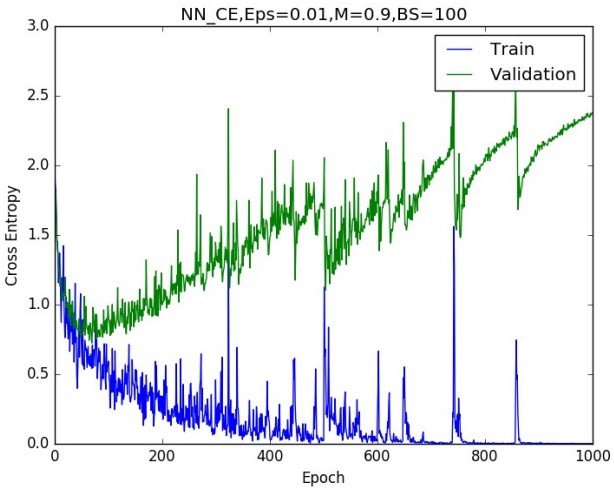


Momentum = 0.5



Momentum = 0.9

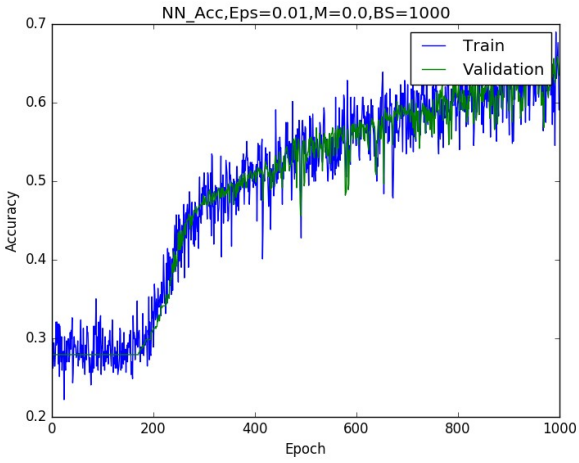


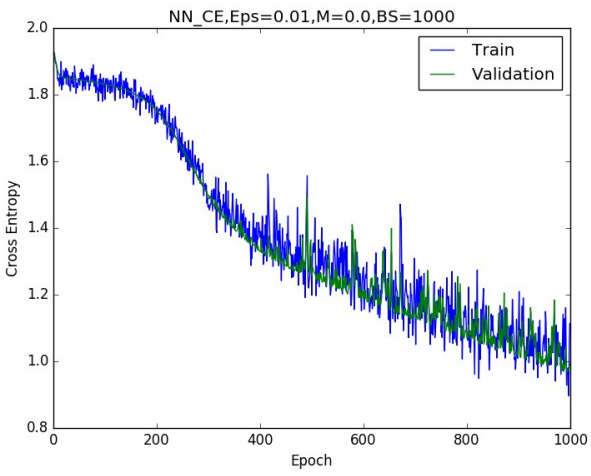


Here it appears that as momentum is increased, the time to convergence decreases for the neural network at the expense of cross entropy. It reaches a peak of accuracy much faster, then it begins to accumulate further loss in the validation set than for the default setting. Accuracy performance seems to be much more stable (and slighty better) for momentum = 0.9 vs the default of 0.0 on the validation set. Cross entropy peaks earlier as momentum increases, but also diverges and begins growing more quickly as well (signalling overfitting is occurring). It would appear that a higher momentum value would help to train the network more quickly, but also implies that training should be stopped sooner here than in the default network.

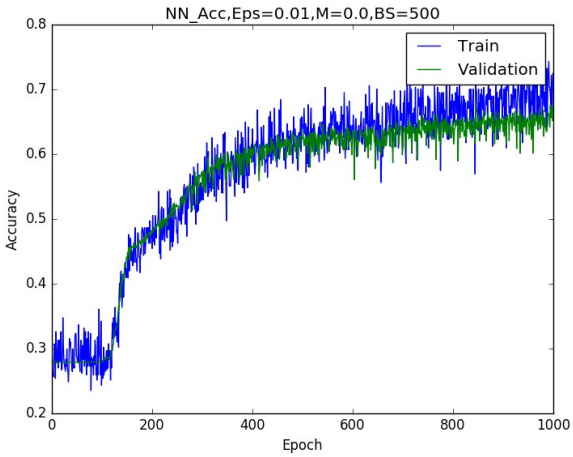
Plots for Batch Size

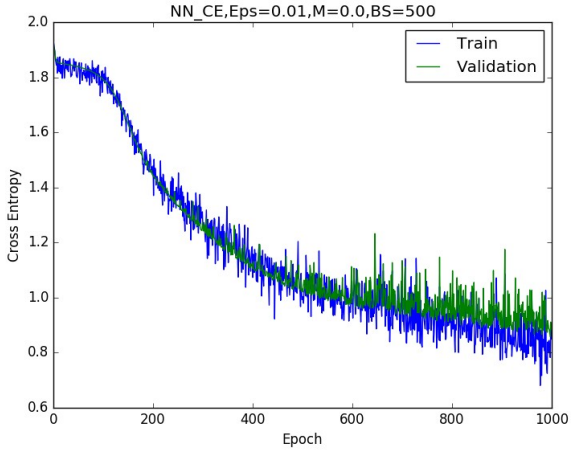
Batch Size = 1000



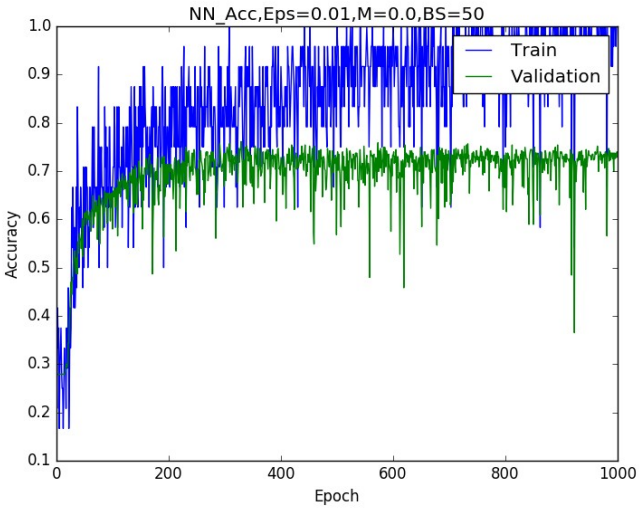


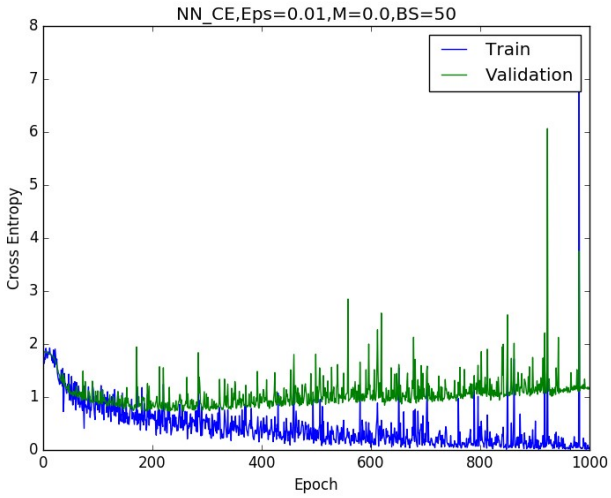
Batch size = 500



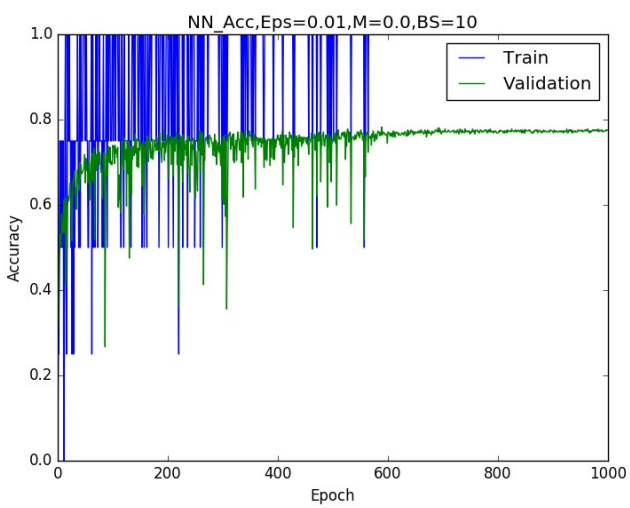


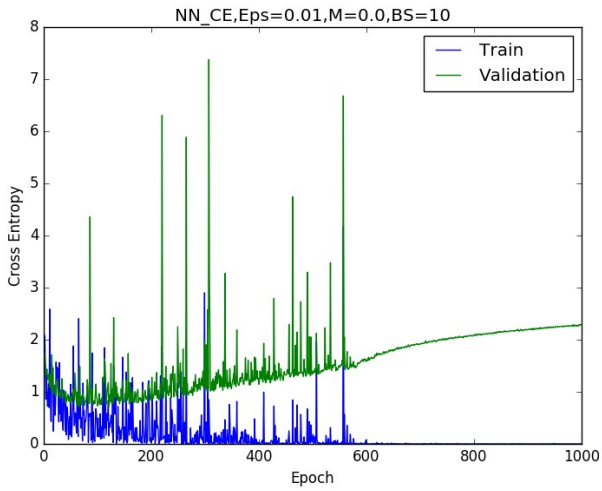
Batch Size = 50



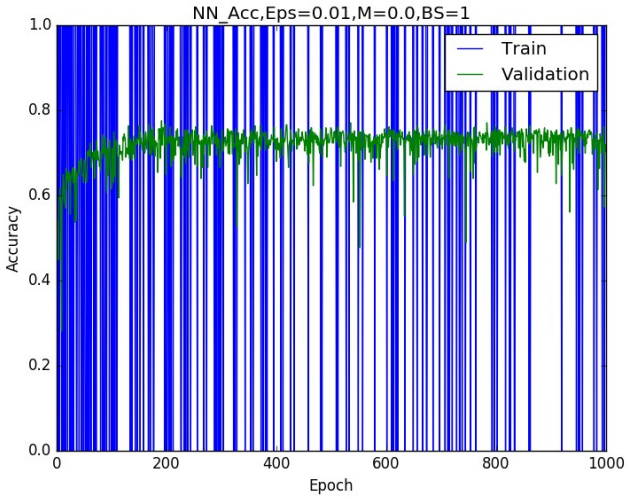


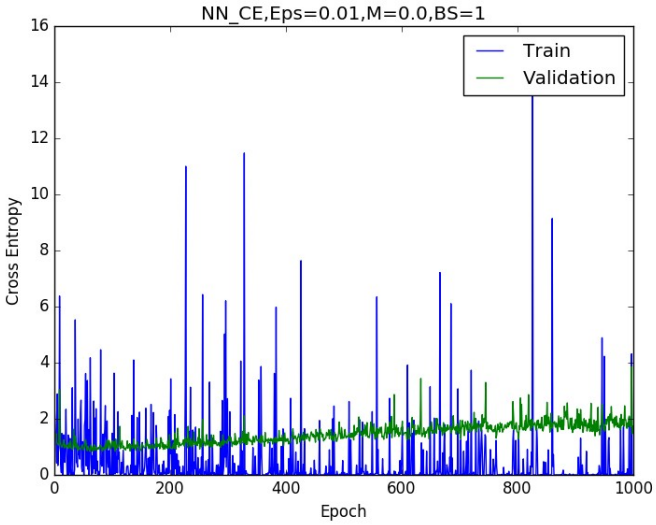
Batch Size = 10





Batch Size = 1





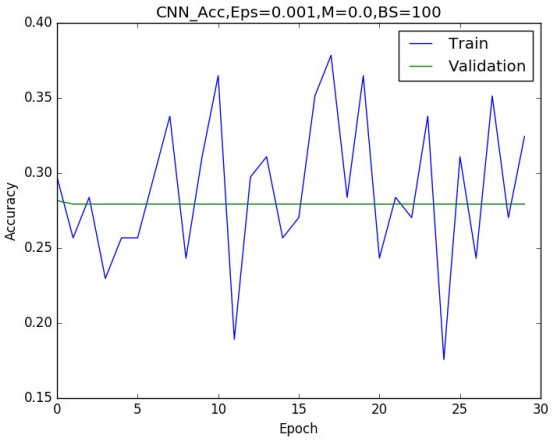
It appears that here, a similar phenomenon is occuring as was with momentum. As the batch size increases, more examples have to be seen before the gradients can be updated. It leads to smoother plots for the training set, as the gradient is not being evaluated until more data points are observed. As batch size increases, the time to convergence on the validation set decreases, in particular it is less than 100 epochs for the stochastic batch size of 1. Conversely, as batch size decreases, the final value for the cross entropy increases, although it reaches a minimum value faster than in the default setup. Here, one can note that the accuracy and cross entropy do not seem to converge for a batch size of 1000, but converge faster and faster on the validation set as the size decreases. The accuracy seems to reach the same maximum value on the validation set for different setups, so the only gain here is again in convergence rate, at the expense of cross entropy.

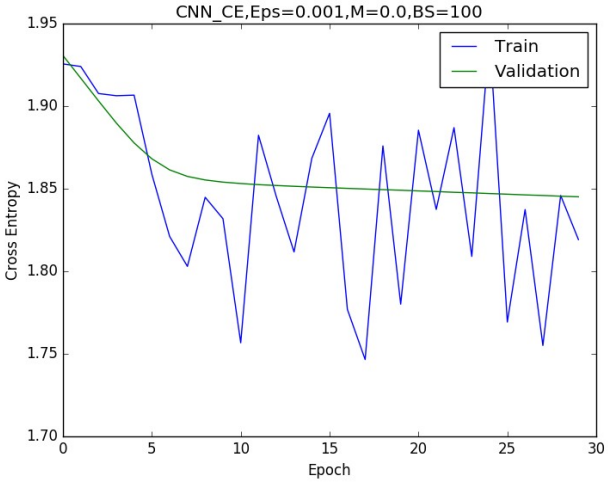
Convolutional Neural Network

For my experiments with hyperparameters, I chose epsilon values of (0.001, 0.05, 0.01, 0.5, 1.0), momentum values of (0, 0.5, 0.9) and batch sizes of (1, 10, 50, 500, 1000). While varying one parameter, I kept the others at their default values. Here are the relevant plots:

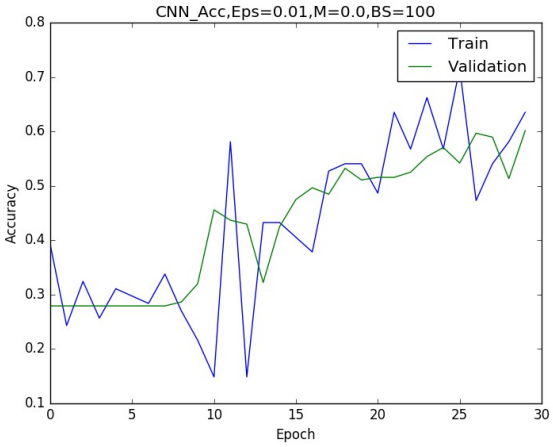
Plots for Epsilon

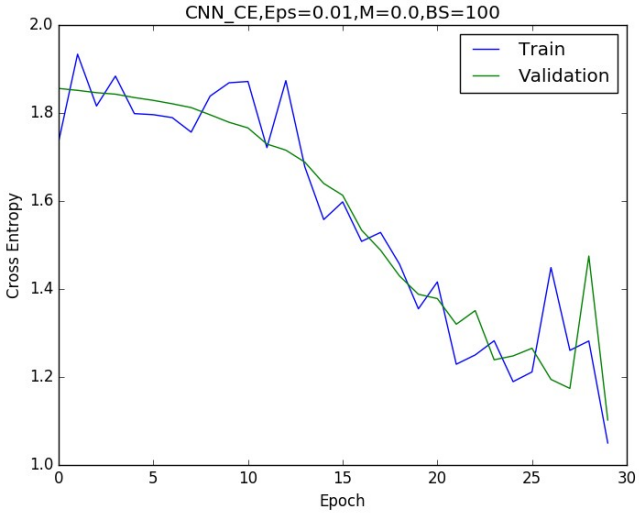
Epsilon = 0.001



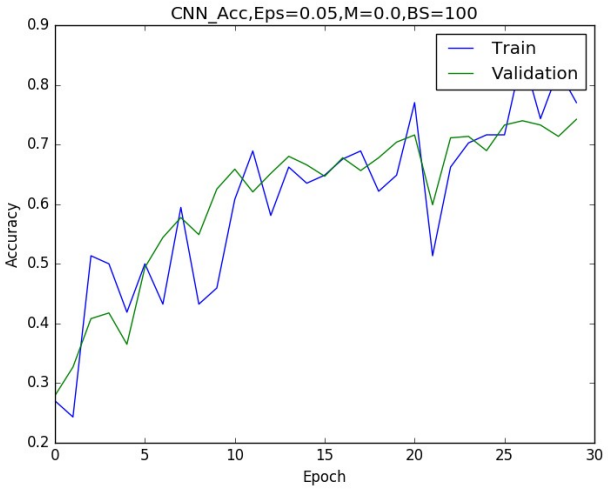


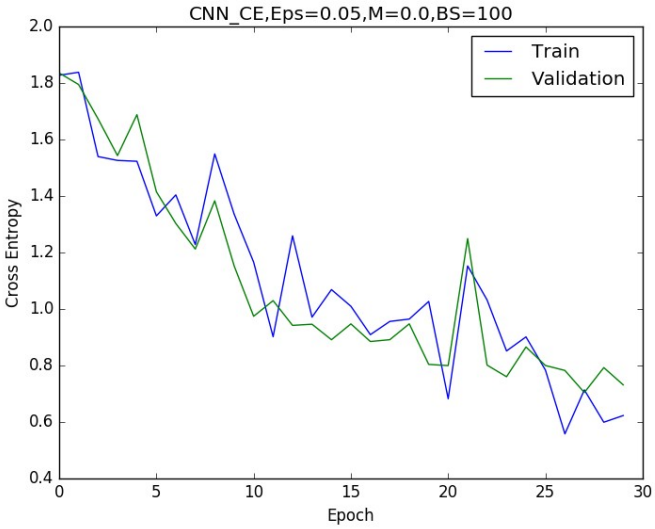
Epsilon = 0.01



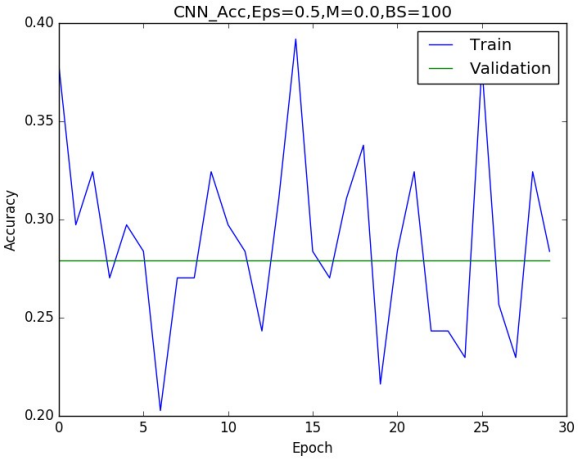


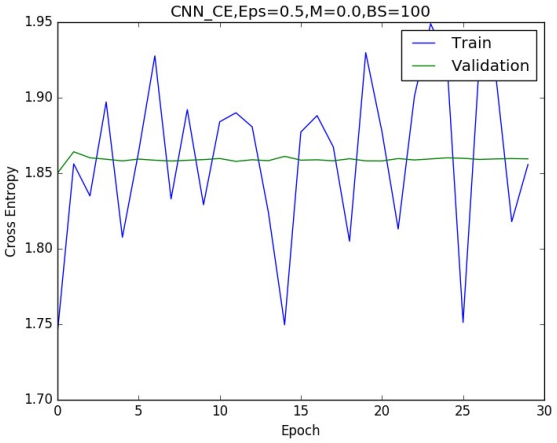
Epsilon = 0.05



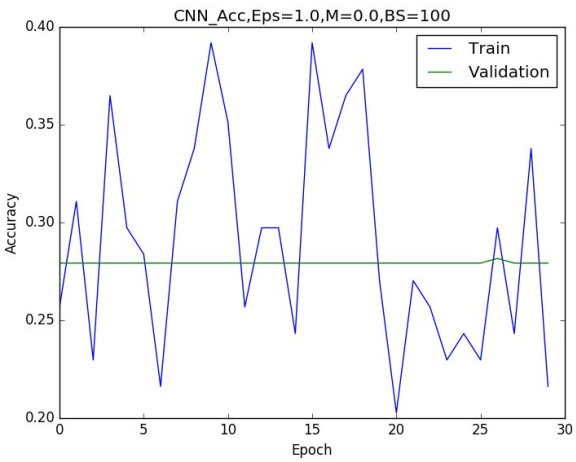


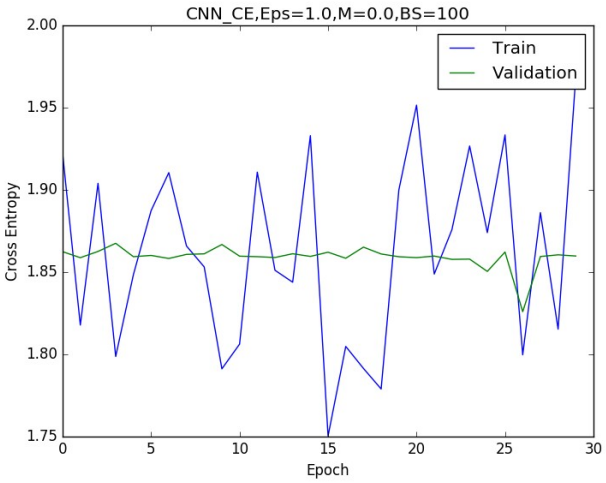
Epsilon = 0.5





Epsilon = 1.0

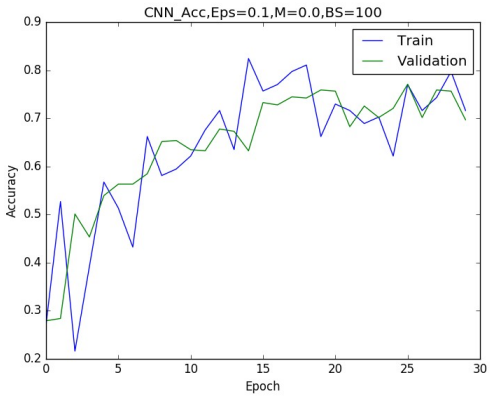


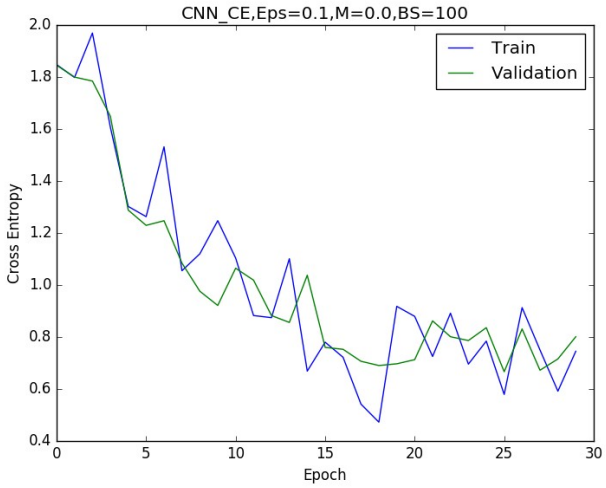


Here it appears that the network is much more sensitive to changes in epsilon than the neural network. With an epsilon that is too high or too low, it fails to converge. At epsilon= 0.001 it fails to move at all, suggesting no learning takes place. As epsilon increases between 0.001 to 0.1 however, it converges more quickly. In particular, it seems that both accuracy and cross entropy for the validaiton set converge at epoch 25 for epsilon = 0.01, at epoch 20 for epsilon = 0.05, and at epoch 15 for epsilon =0.1. In all of these cases, the validaiton set performance is similar, but it seems to reach the highest accurracy and lowest cross entropy values for epsilon = 0.05 (even better than the default hyperparameter settings). With epsilon values of 0.5 and beyond however, it fails to converge, suggesting the learning rate is far too high in these cases.

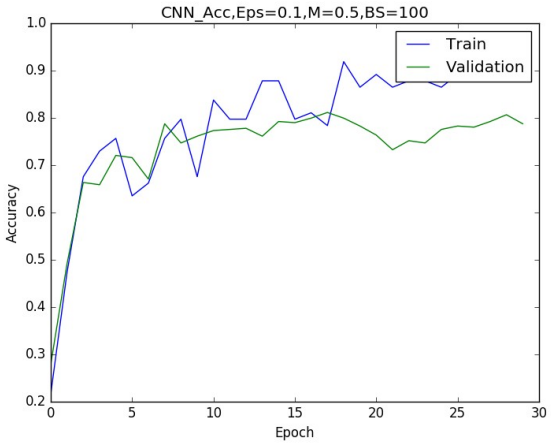
Plots of Momentum

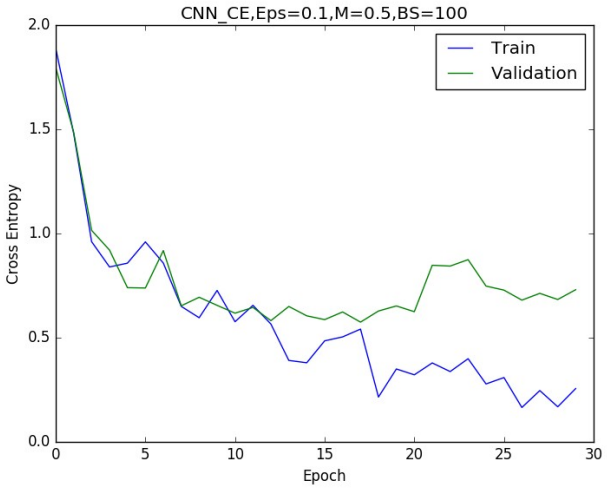
Momentum = 0



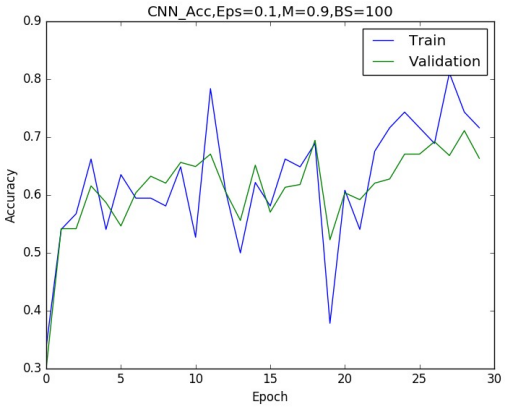


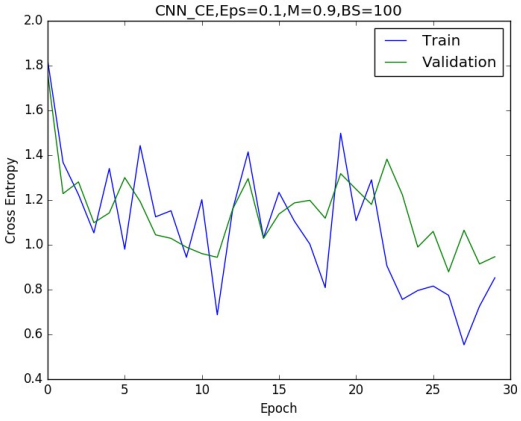
Momentum = 0.5





Momentum = 0.9

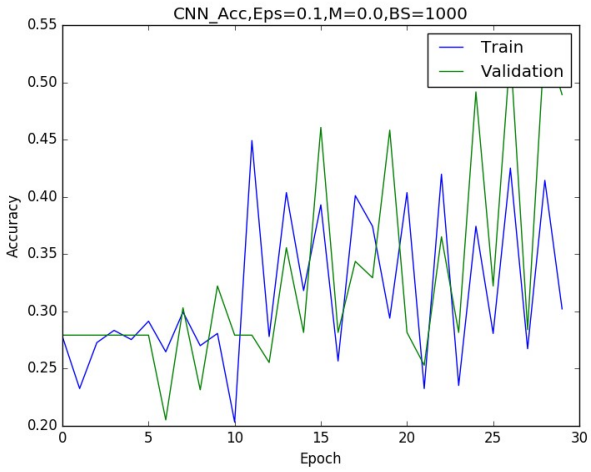


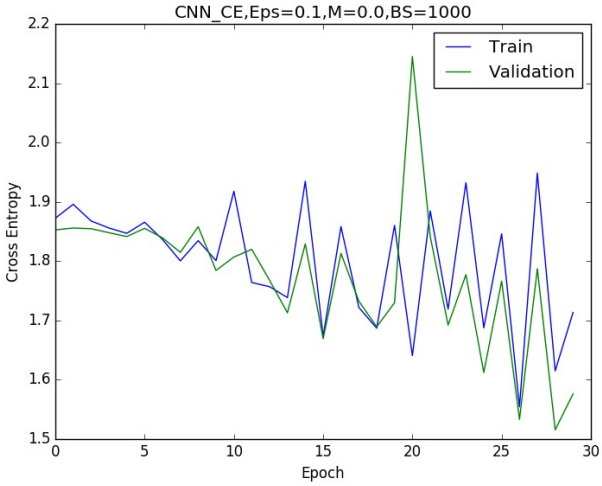


Like with the neural network, as the momentum increases, the network seems to converge more quickly, although again at the expense of cross entropy. In the default setting of momentum=0, it takes around 20 epochs for the network to converge. With momentum=0.5 it converges after around 8 epochs and with momentum=0.9 it converges after 6 epochs. As with the neural network however, the cross entropy for the validation set increases for the validation set as momentum is increased. The overall performance seems to be best for momentum=0.5, although it may be overfitting the training set by the final few epochs as the accuracy on the training set is approaching 1.0.

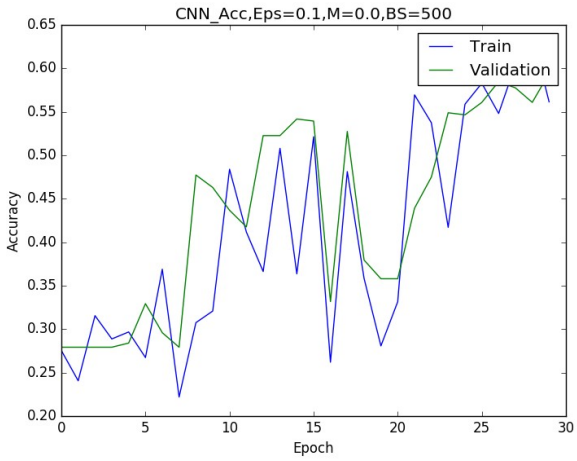
Plots of Batch Size

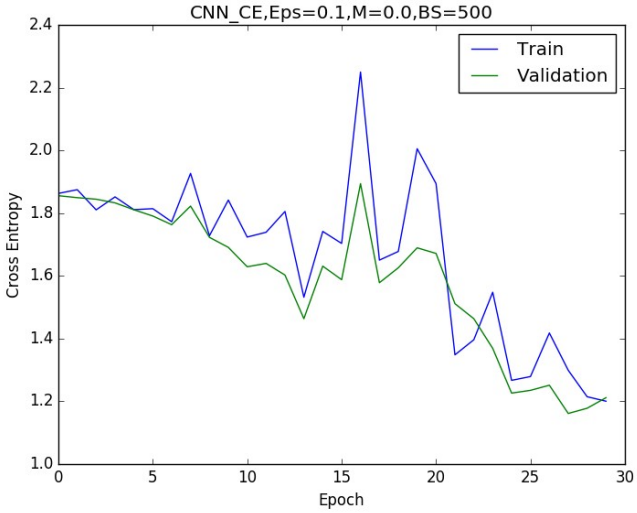
Batch Size = 1000



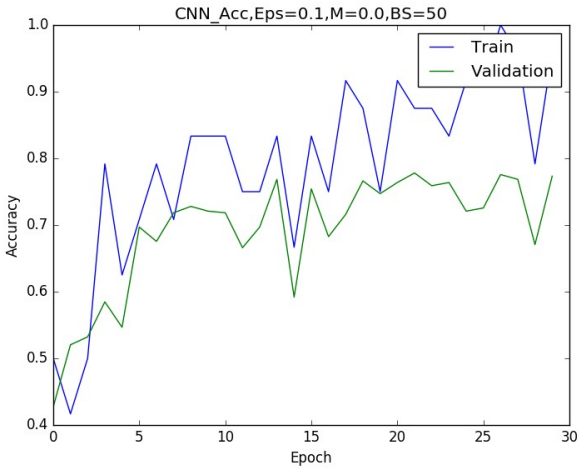


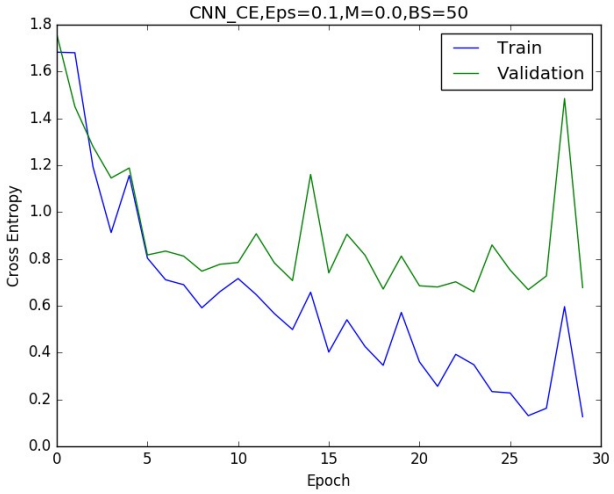
Batch Size = 500



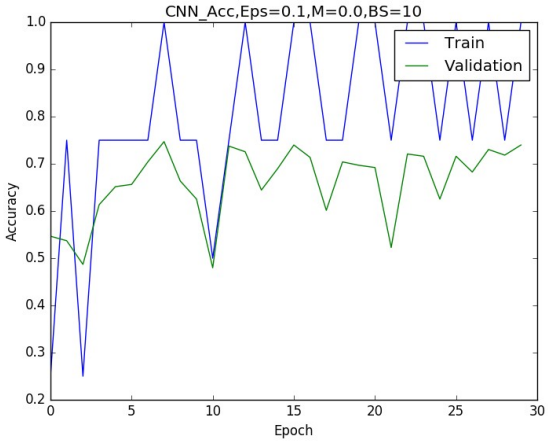


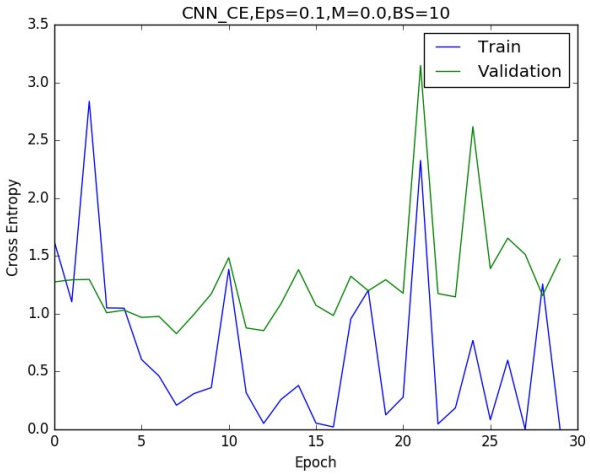
Batch Size = 50



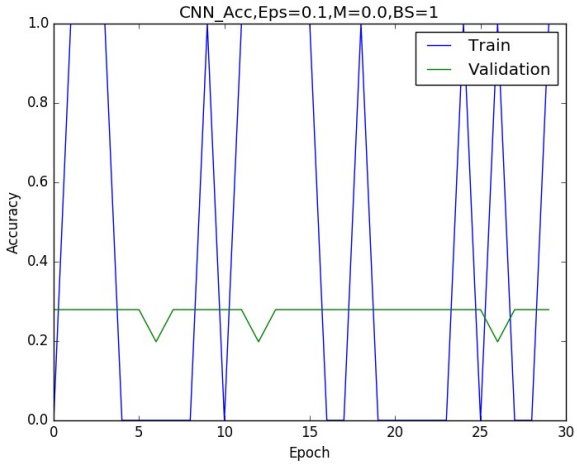


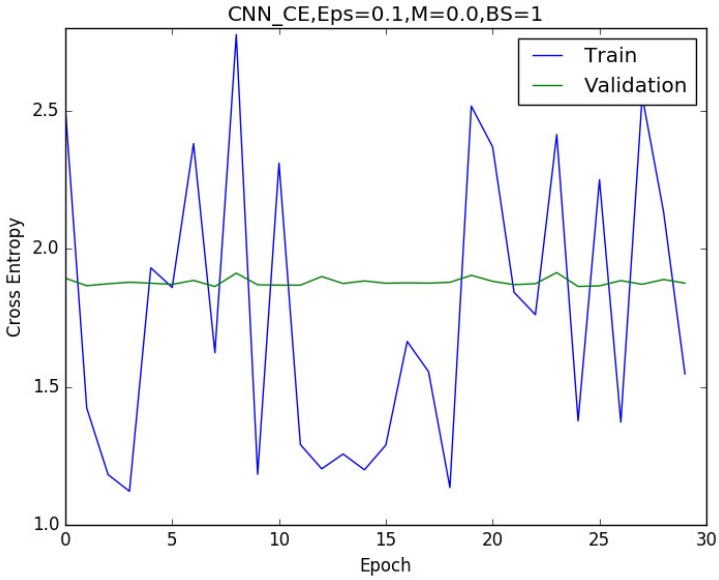
Batch Size = 10





Batch Size = 1.0





Like with epsilon, there are some Goldilocks-tendancies here too. With a very large batch size the network does not converge as it must view many more data points before updating its weights. For very large batch sizes the network appears to perform worse than the default network, but given further epochs, may eventually converge to a better accurracy and cross entropy value. As the batch size increases to 10 and 50, the network converges to its maximum accuracy and minimum cross entropy values for the validation set at epochs 6 and 8 respectively. In both of these cases the networks perform similarly to the default settings, although as the batch size decreases, the cross entropy values appear to increase too. For a batch size of 1, the plot is very noisy and interestingly, the network does not appear to learn at all. This suggests that a batch size of 10-100 works best here.

Finally, to find the best combination of hyperparameters, some type of Grid Search could be implemented, with Cross-Validation. In our case, this would be very excessive (there would be 75 models to consider!), but it would be the best way to get an accurate idea of the best combination of hyperparameters. In our case, observing trends in each hyperparameter would likely be sufficient to narrow down the best combination of hyperparameters for the best results/fastest convergence.

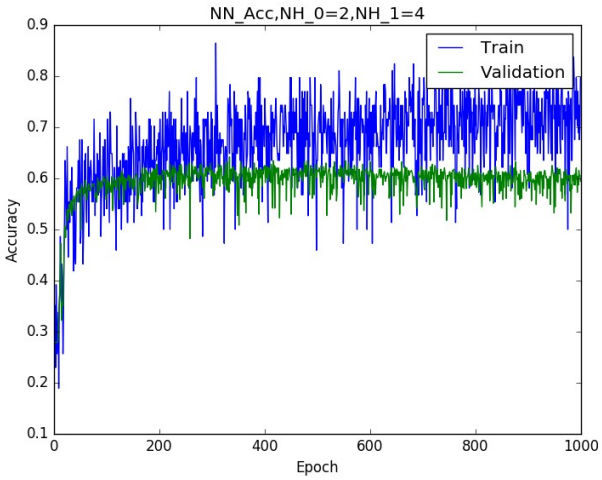
3.3

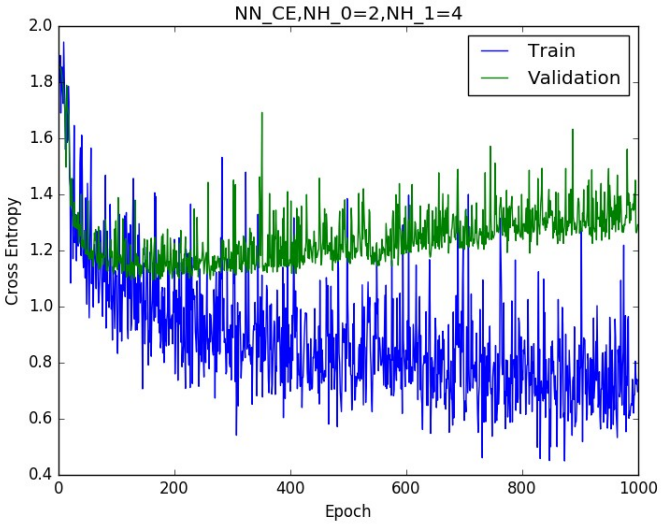
Neural Network

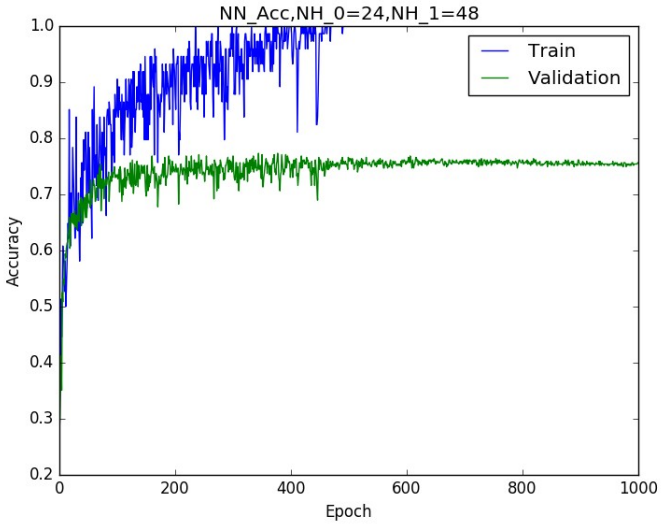
For the Neural Network, I've chosen the hidden unit combinations as follows:

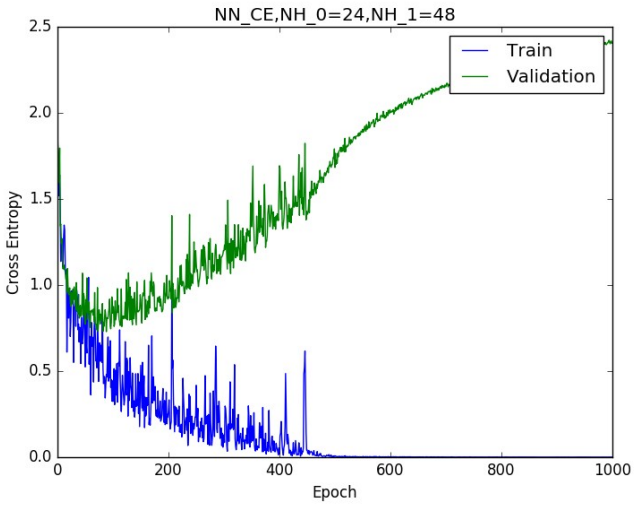
(2,4) , (24,48), (50,100)

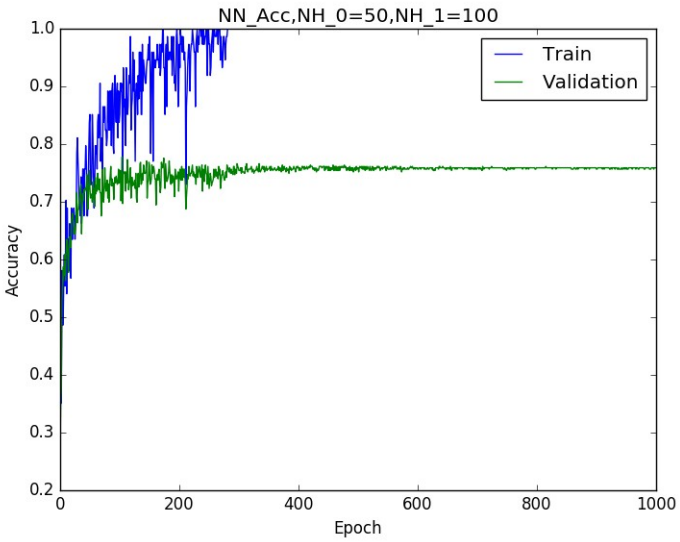
For the hyperparameters, I set momentum to be 0.9 and I left the other hyperparameters as their default values (epsilon = 0.01, batch_size=100). Here are their respective results:

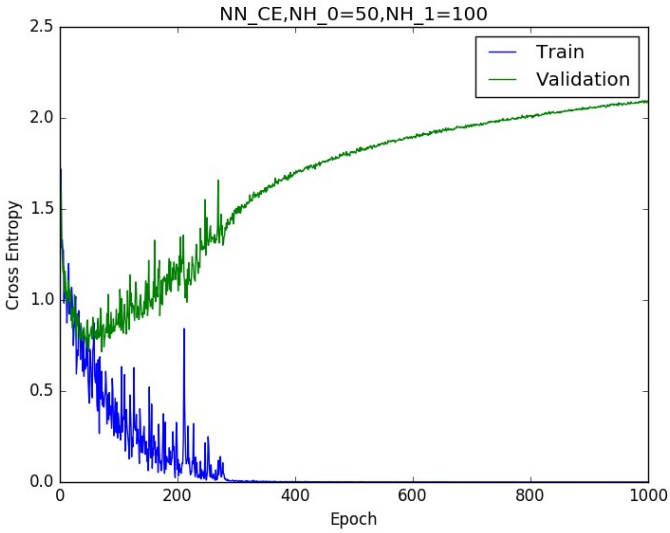












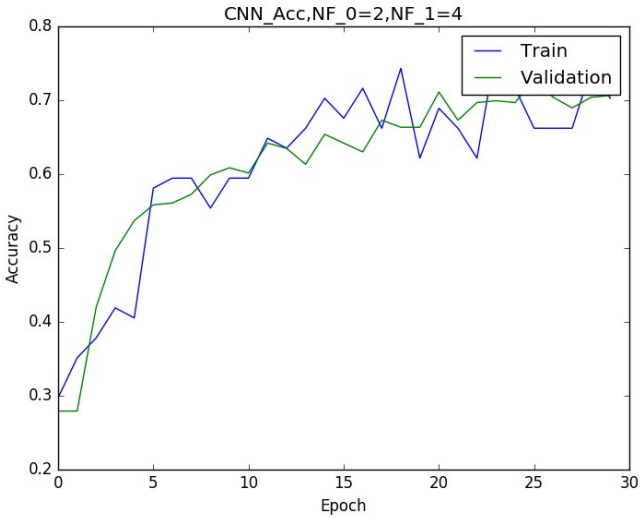
Looking at these plots, one can infer that as the number of hidden layers increases, the network converges more quickly, while overfitting increases. For instance, with 50 and 100 hidden units in the two layers, the network converges after around 300 epochs, but also begins to overfit after less than 100. The overall accuracy is comparable to the that of the default neural network in this case, but the generalization is much worse (as the network obtains 100% accurracy on the training data very quickly and actually begins to increase in cross entropy error in the validation data concurrently). With 24 and 48 hidden units in the two layers the network converges after around 500 epochs, but also beings overfitting within 100 epochs. Finally, for 2 and 4 hidden units, the network is much less accurate, but does not overfit. It also seems to converge relatively quickly, but is far noisier than for the networks with more hidden units. This likely has to do with the capacity of the network. It overfits less than even the default network, but is also far noisier in its accuracies and cross entropy plots. The tradeoff is in performance however, as it is both less accurate and has more error than the default network as it is a far simpler model.

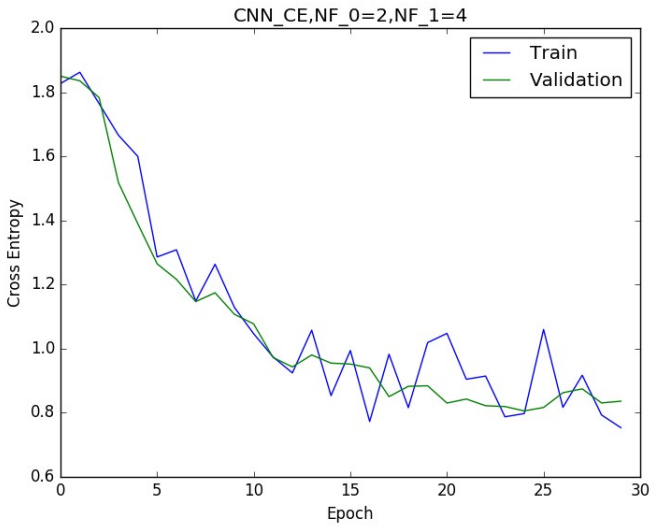
Convolutional Neural Network

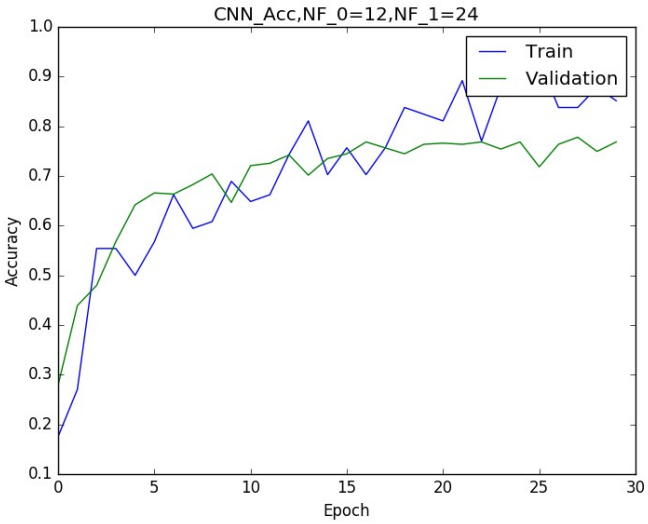
For the Convolutional Neural Network, I've chosen the filter number combinations as follows:

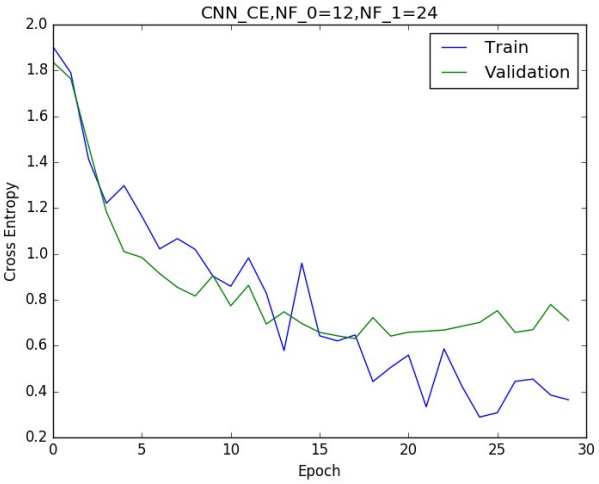
(2,4) , (12,24), (25,50)

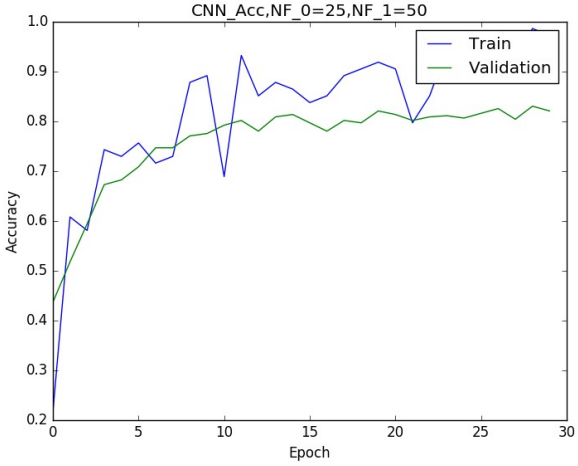
For the hyperparameters, I set momentum to be 0.9 and epsilon to be 0.01, with a batch_size of 100. Here are their respective results:

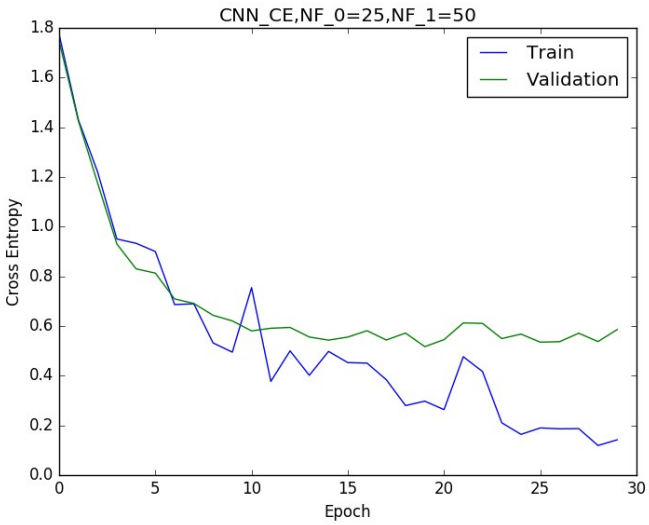

CNN_Acc,NF_0=2,NF_1=4
Train
Validation
Accuracy
Epoch


CNN_CE,NF_0=2,NF_1=4
Train
Validation
Cross Entropy
Epoch


CNN_Acc,NF_0=12,NF_1=24
Train
Validation
Accuracy
Epoch


CNN_CE,NF_0=12,NF_1=24
Train
Validation
Cross Entropy
Epoch


CNN_Acc,NF_0=25,NF_1=50
Train
Validation
Accuracy
Epoch


CNN_CE,NF_0=25,NF_1=50
Train
Validation
Cross Entropy
Epoch

Similar to with the neural network, the convolutional neural network seems to converge faster, but unlike the neural networks above, don't seem to overfit as severely with a higher number of filters. I suspect the latter is avoided as there are far fewer epochs for these networks, leading to less opportunity for overfitting to occur. With 25 and 50 for the filter combination, the network converges after about 10 epochs. With 12 and 24 filters, the network converges after about 15 epochs. Finally, with 2 and 4 filters, the network converges after around 25 epochs, but has not only worse accurracy, but higher loss and less accuracy than the default convolutional network. The networks with (25,50) filters actually performs marginally better than the default convolutional neural network (it is more accurate and has less error) and converges faster. The network with (12, 24) filters performs similarly to the default network, but appears to be slightly more accurate (but again, not by much). It takes longer to train networks with a higher number of filters, which appears to be the only tradeoff in this case with performance versus the default parameters.

3.4

Neural Network:

In a Neural Network, the number of parameters between two successive layers is defined as:

*num_param = (input_dimension * output_dimension) + output_dimension*

where the final addition above comes from the additional biases.

So, for our default Neural Network, the total number of parameters becomes:

(num_inputs * num_hiddens[0]) + num_hiddens[0] + (num_hiddens[0] * num_hiddens[1]) + num_hiddens[1] + (num_hiddens[1] * num_outputs) + num_outputs

= (2304 * 16) + 16 + (16 * 32) + 32 + (37 * 7) + 7 = 37655

Convolutional Neural Network:

In a Convolutional Neural Network, the number of parameters for a convolutional layer is defined as:

*cl_num_param = (filter_size * filter_size * input_depth) * num_filters + num_filters*

where the final addition above comes from the additional biases.

There are no additional parameters that come from the pooling layer.

The fully connected layer contributes an additional number of parameters defined as follows:

fcl_num_param = (input_dimension * output_dimension) + output_dimension

So for our default network of two convolutional layers and one fully connected layer, we have the following:

total_params = cl_1_num_param + cl_2_num_param + fcl_num_param =

(filter_size * filter_size * num_channels) * num_filters_1 + num_filters_1 + (filter_size * filter_size * num_filters_1)* num_filters_2 + num_filters_2 + (num_filters_2 *64 * num_outputs) + num_outputs

= (5 * 5 * 1) * 8 + 8 + (5 * 5 * 8) * 16 + 16 + (16 * 64 * 7) + 7 = 208 + 3216 + 7175 = 10599

*Note: The * 64 above comes from the need to transform a 4D tensor to a 2D tensor in the connection from the convolutional layer to the fully connected layer's affine layer.*

In any case, the Neural Network, as expected has more parameters (almost 4 times more!)

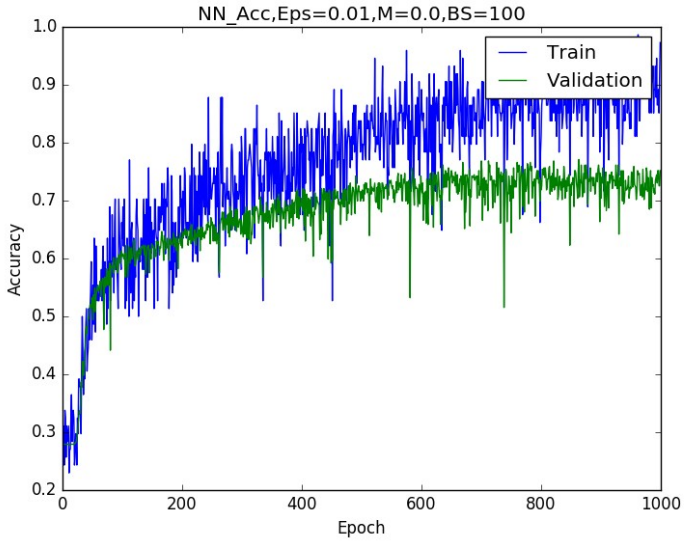
To get the parameters to be similar between the two networks, I choose to keep the parameters fixed in the Neural Network, and increase them in the Convolutional Neural Network. Taking num_filters_1 = 20 and num_filters =40 (rather than the default 8 and 16 respectively) one gets the following:

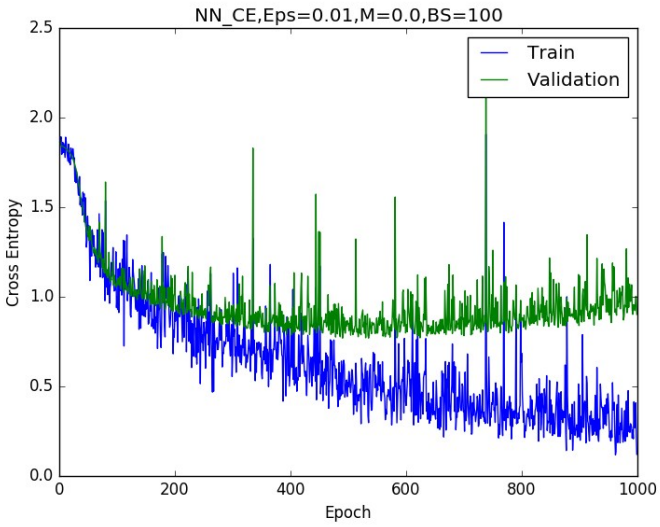
modified_cnn_params = (5 * 5 * 1) * 20 + 20 + (5 * 5 * 20) * 40 + 40 + (40 * 64 * 7) + 7 = 520 + 20560 + 17927 = 39007

As abs(37655-39007)/37655 ~ 0.036 = 3.6%, this is very similar in scale to the default neural network for parameters.

Here is a comparison of their performance:

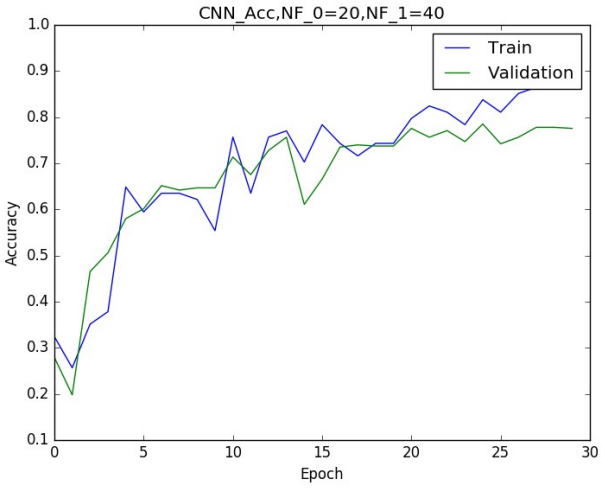
Default Neural Network

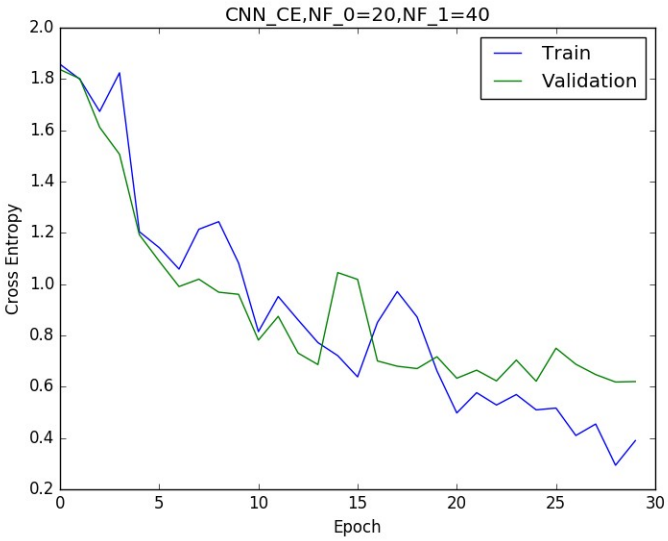




CE: Train 0.37885 Validation 1.06263 Test 0.97500
Acc: Train 0.85507 Validation 0.73031 Test 0.71688

Convolutional Neural Network (with 20, 40 for filter numbers)



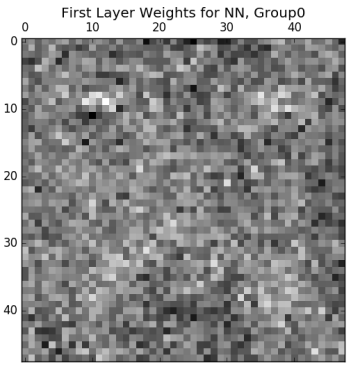


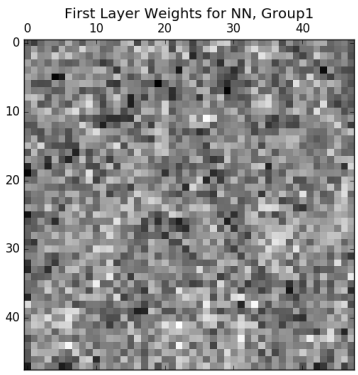
CE: Train 0.30417 Validation 0.62007 Test 0.57882
Acc: Train 0.90190 Validation 0.77566 Test 0.80260

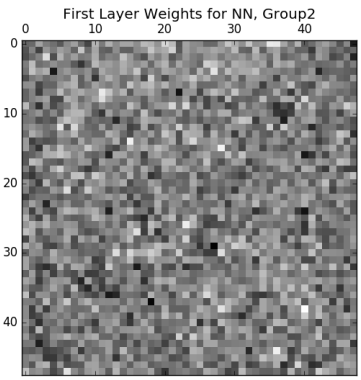
Unsurprisingly, the CNN obtains better generalization. It is more accurate by 10% on the test set and has half the cross entropy error. This follows as CNN's are constructed primarily to deal with image recognition tasks, and by sharing weights and examining local areas, they exploit the spatial structure of images far better than basic neural networks can. By nearly quadrupling the number of parameters in the CNN, I essentially just added almost 3 times more filters. This allows the network to learn even more complicated features, expanding its already intrinsic advantage over the basic neural network. Consequently it has better performance.

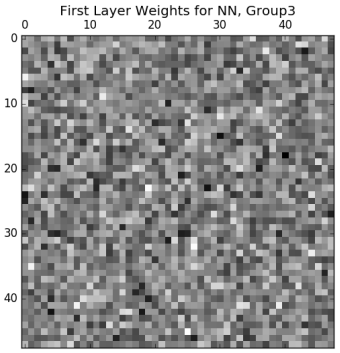
In order to better plot the first layer weights of the Neural Network, I have reshaped the matrix from 2304 * 16 to be (48 *48) * 16 instead. They are plotted below.

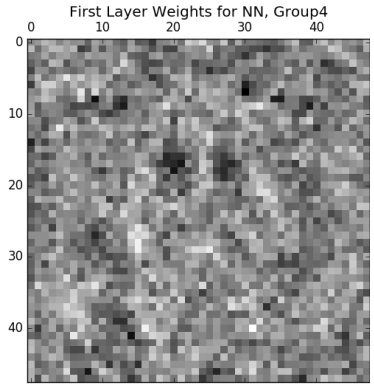
First Layer Weights

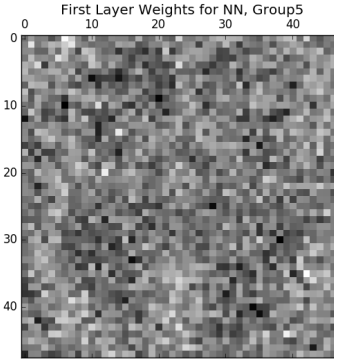


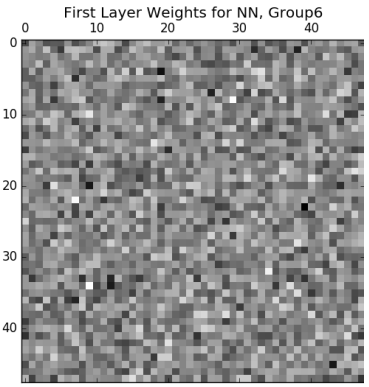


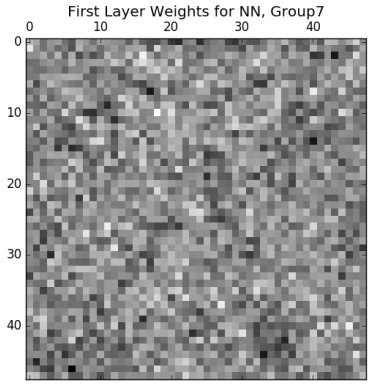


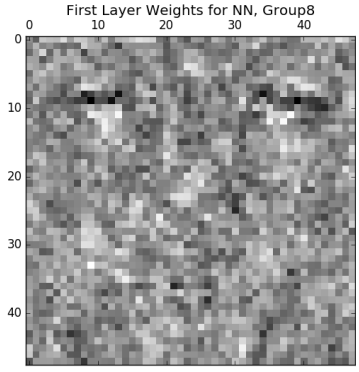


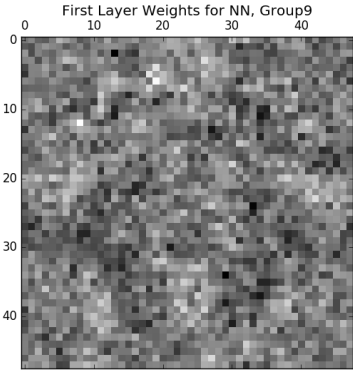


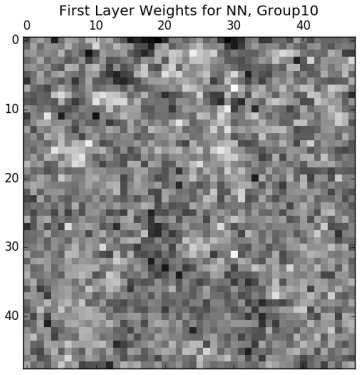


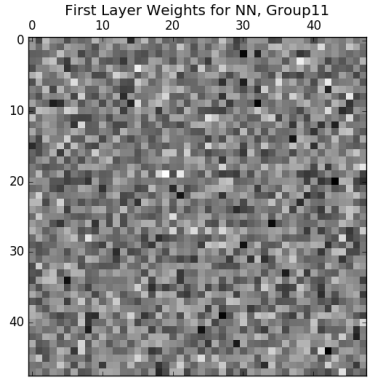


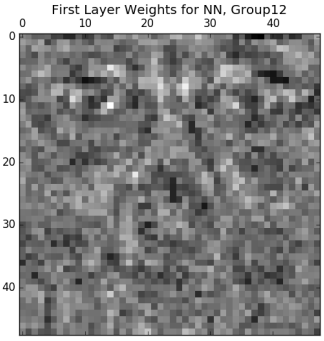


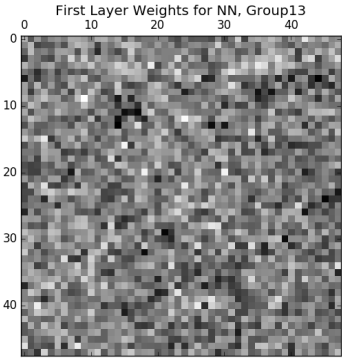


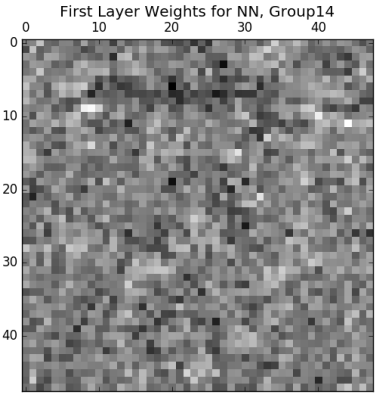


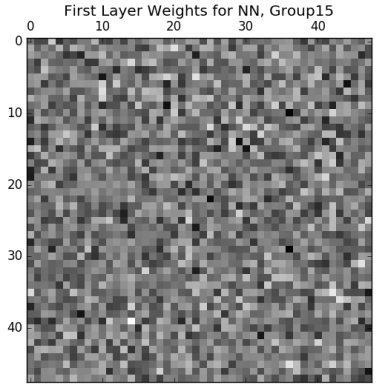








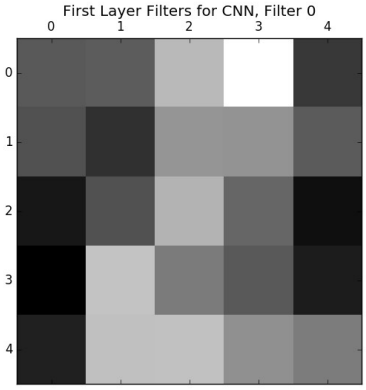


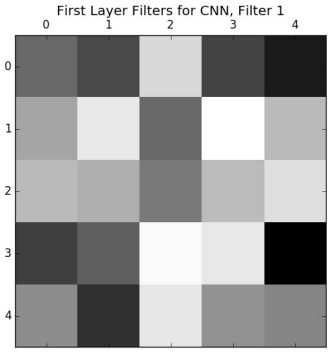


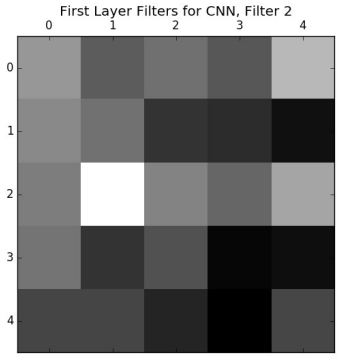
Looking at these first layer weights for the neural network, one can see they are abstract visualizations of facial features. In particular, groups 0,8,9,10 and 12 look like creepy faces if you squint hard enough. It is difficult to explain what each weight is learning/modelling specifically, but it appears that some are looking at things like eyes/eye shape, mouth position, etc... In any case, it’s an interesting visualization!

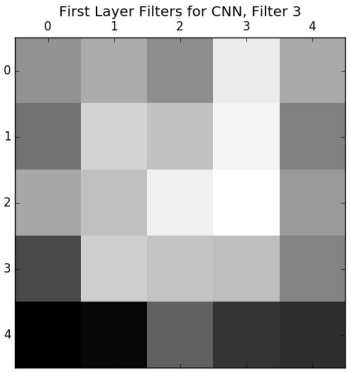
First Layer Filters of the CNN

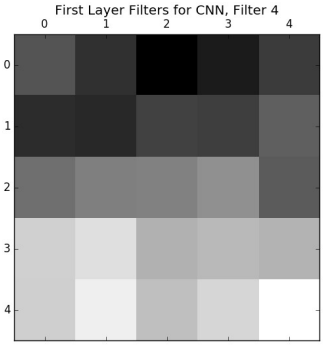
Here are plots of the 20 First Layer Filters of the CNN:

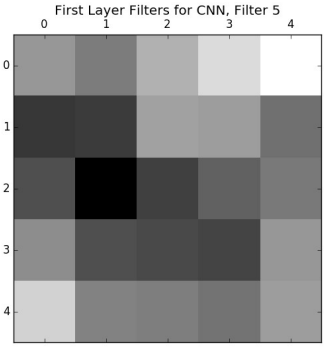


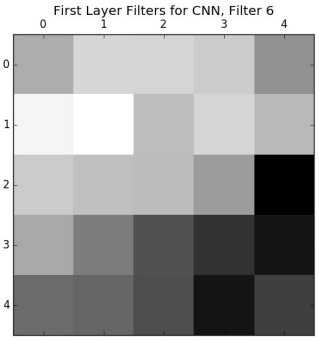


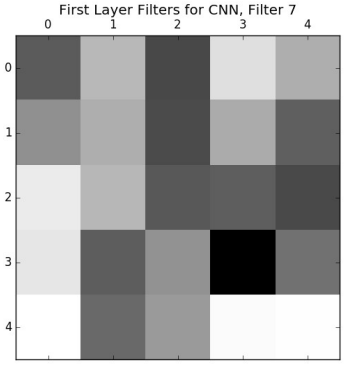


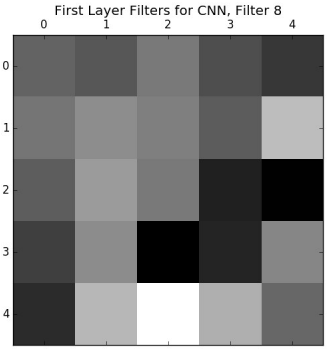


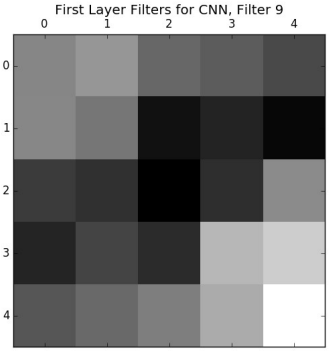


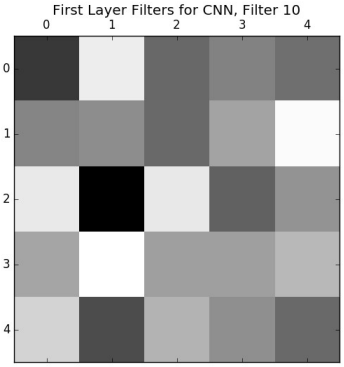


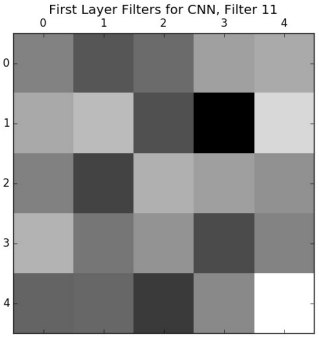


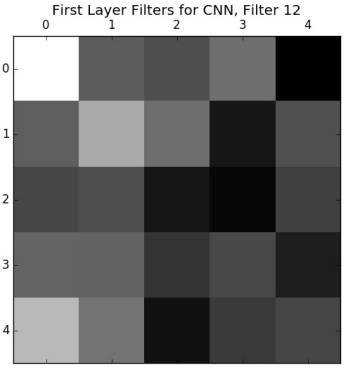


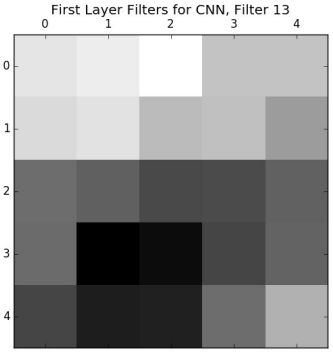


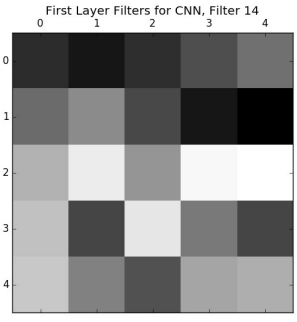


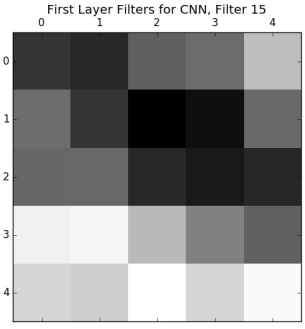


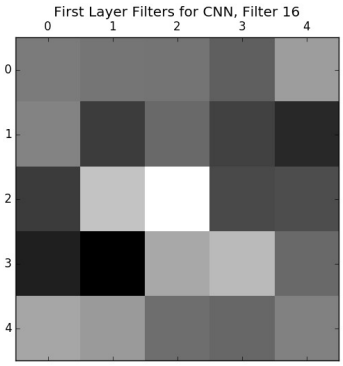


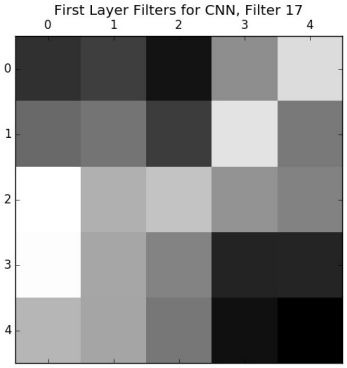


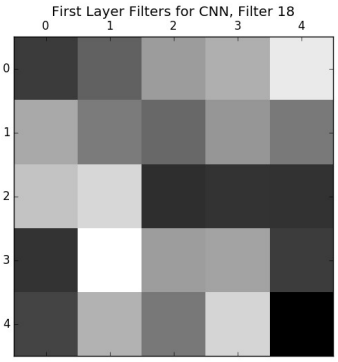


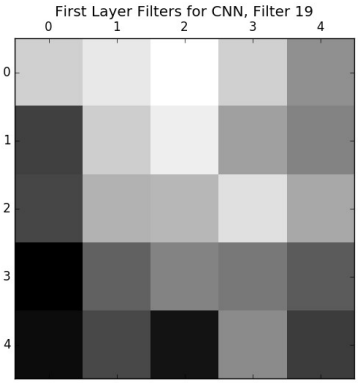










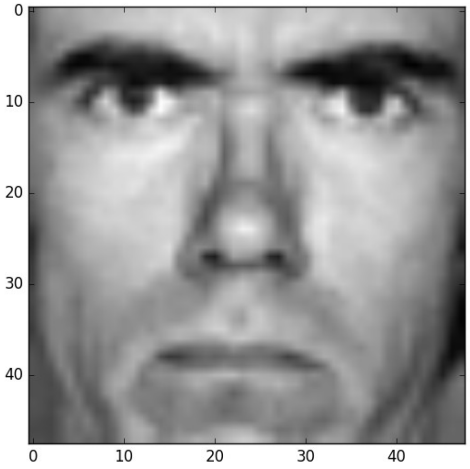


For the first layer filters, it appears that the network is learning to identify things like light and dark regions of the images. In the case of faces, this could mean it is observing were shadows tend to occur on faces (e.g. the nose, beneath eyebrows, etc...). In subsequent layers it will learn more complicated information like edges, complicated shapes and actual facial features by expanding on this initial interpretation.

3.5

I set a threshold of (2/7) for network uncertainty. If the highest score had a probability lower than this, I saved the example, its probability and its prediction. For both networks, I used the default hyperparamters. Here are the plots:

Default Neural Network



Above is image 170 in the validation set. The network predicted with softmax score 0.2774 that this face was neutral. The correct target is in fact angry (this was the second highest softmax score which it

predicted with probability 0.2773). In this case, the network is definitely not confident and unfortunately got the classification incorrect by outputing the highest score.

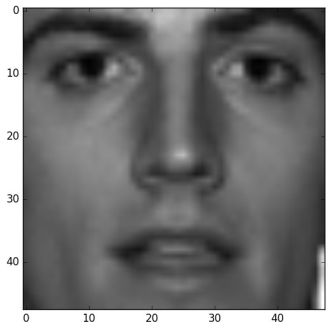
Full softmax output for this face:

[0.27735642 0.00162479 0.16682891 0.00140601 0.19614639 0.07920172
  0.27743576]

So here it was really torn between two choices.

Default Convolutional Neural Network

Training Set

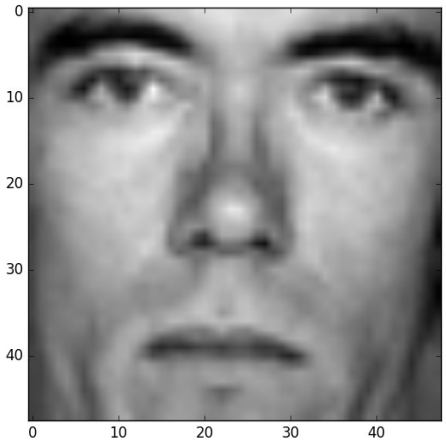


Above is image 173 from the
training set. It thought the expression was angry with a softmax score of
0.24. Here the correct target is neutral, which it predicted with a softmax score of 0.205. In other words, this is actually very similar to the uncertainty above in the neural network, just with the emotions reversed. Here again, the network was not confident and actually made the wrong prediction between its two highest options.

Full softmax output for this face:

[0.23994943 0.15444303 0.05461541 0.16983587 0.15332245 0.02243239
  0.20540143]

Validation Set

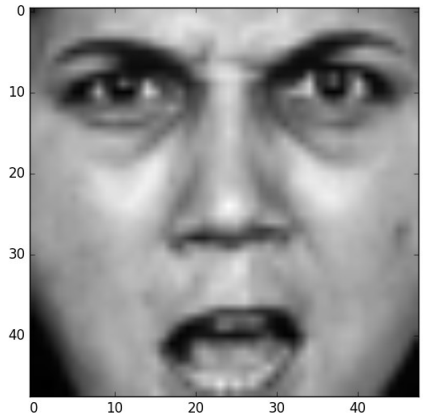


Here we have the same person as in the neural network inaccuracy, which makes sense as this is image 173 of the validation set. In this case, the correct expression target is neutral, which the network unfortunately did not predict. Here, like in the training inaccuracy for the CNN, it predicted angry with a softmax score of 0.24. It predicted netural with a softmax score of 0.20. Again, by outputing the highest score, it is not correct in this case. Like in the previous cases, it was torn between two choices.

Here are all the softmax scores for this face:

[0.23994943 0.15444303 0.05461541 0.16983587 0.15332245 0.02243239
  0.20540143]

Test Set



Here we have image 198 of the test set. The network predicted that this face was surprise with a softmax score of 0.280. The correct label for this face is actually angry, which got a softmax score of 0.25. Here again, the network was uncertain as the top scores were all similar, but by going with the highest value, it was incorrect.

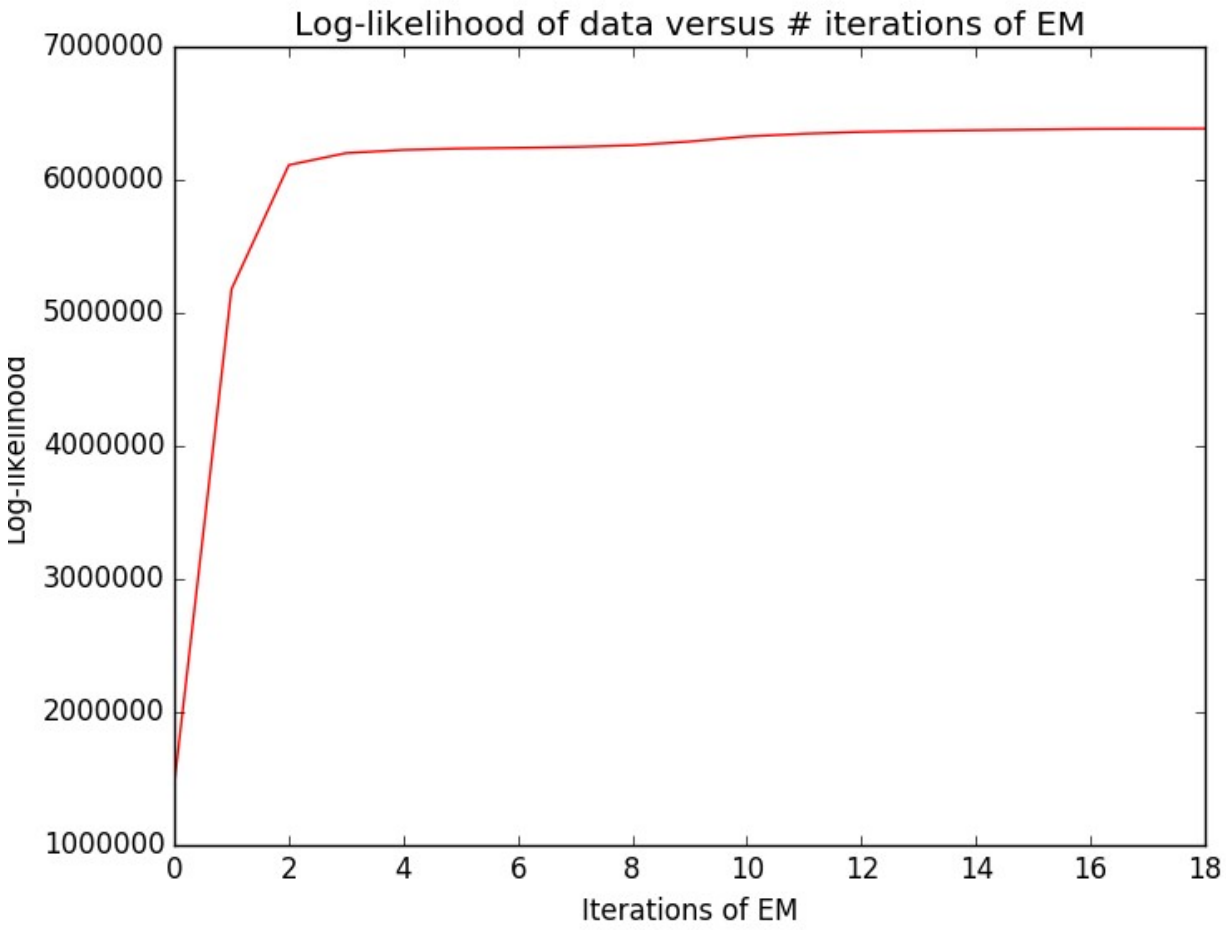
Here is the full softmax output:

[0.25038703 0.19903878 0.20576177 0.00046154 0.05101528 0.28049365
  0.01284195]

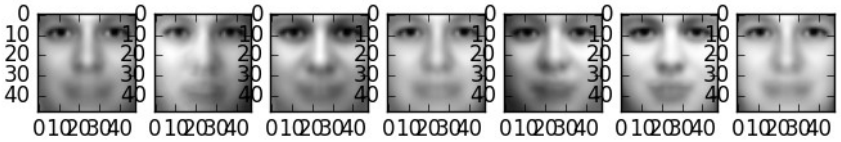
So in conclusion, for all of the uncertainties in my two networks (top scores below 2/7 in value), I found that it actually lead to incorrect results. It seems like the two networks had a hard time telling the difference between angry and neutral faces, and specific people seemed to exacerbate that further (the man in validation 170,173). There could definitely be possibilities where the network is correct, but highly uncertain, I was just unable to find any examples of this with the default hyperparameters.

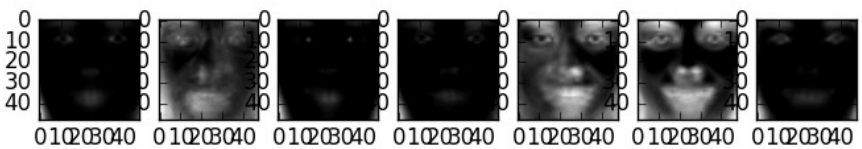
4.2

K = 7
iterations = 20
minVary = 0.01
randConst = 10
Iter 20 logLikelihood 6363465.17084



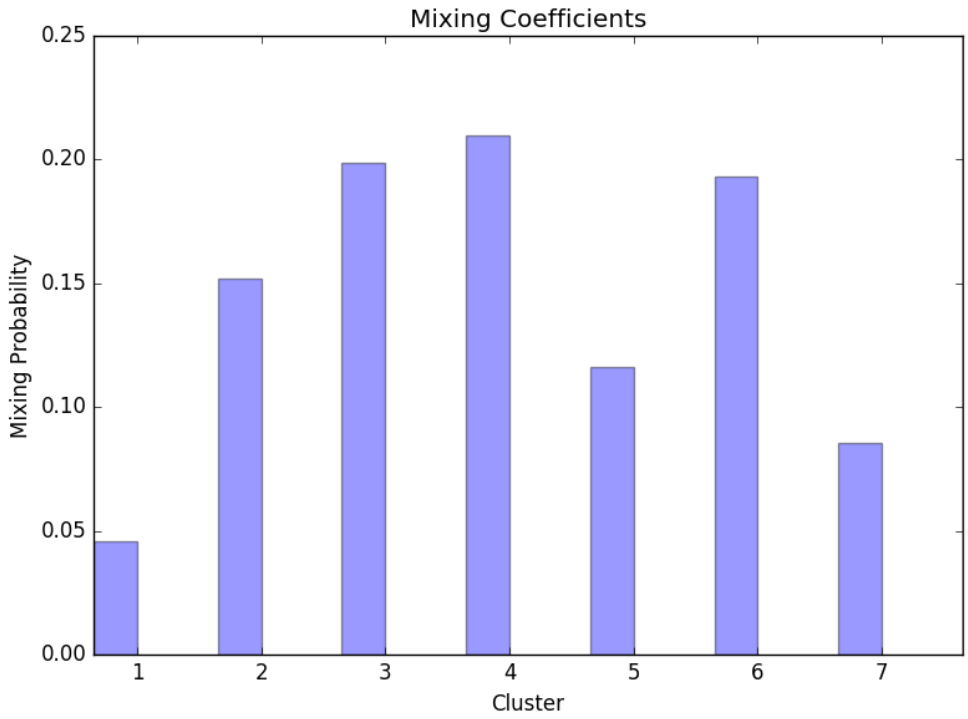
Mean Vectors for Each Cluster





Variance Vectors for Each Cluster

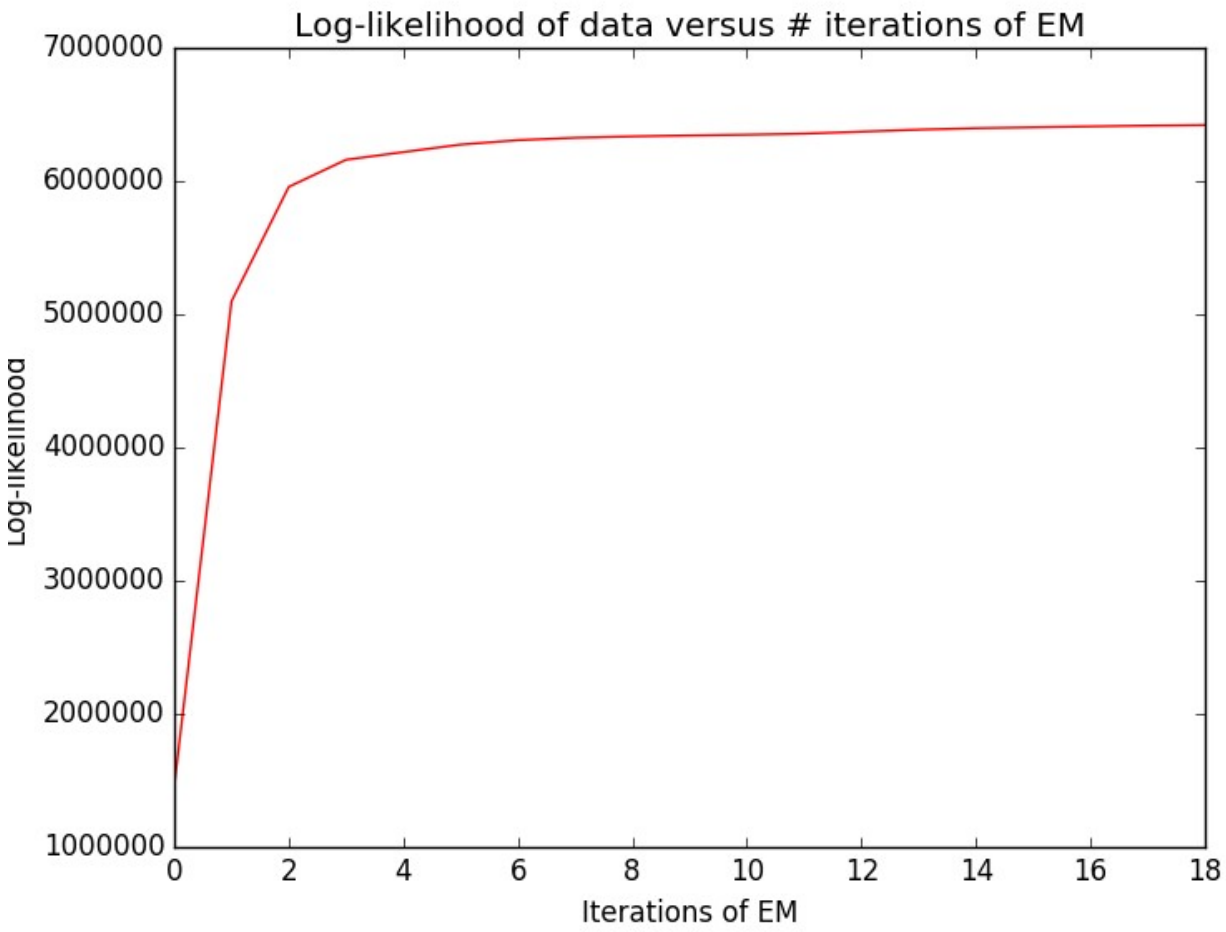
mixing coefficients = [[0.04564315]
 [0.15180142]
 [0.19850785]
 [0.2094946]
 [0.11595867]
 [0.193236]
 [0.0853583]]



4.3

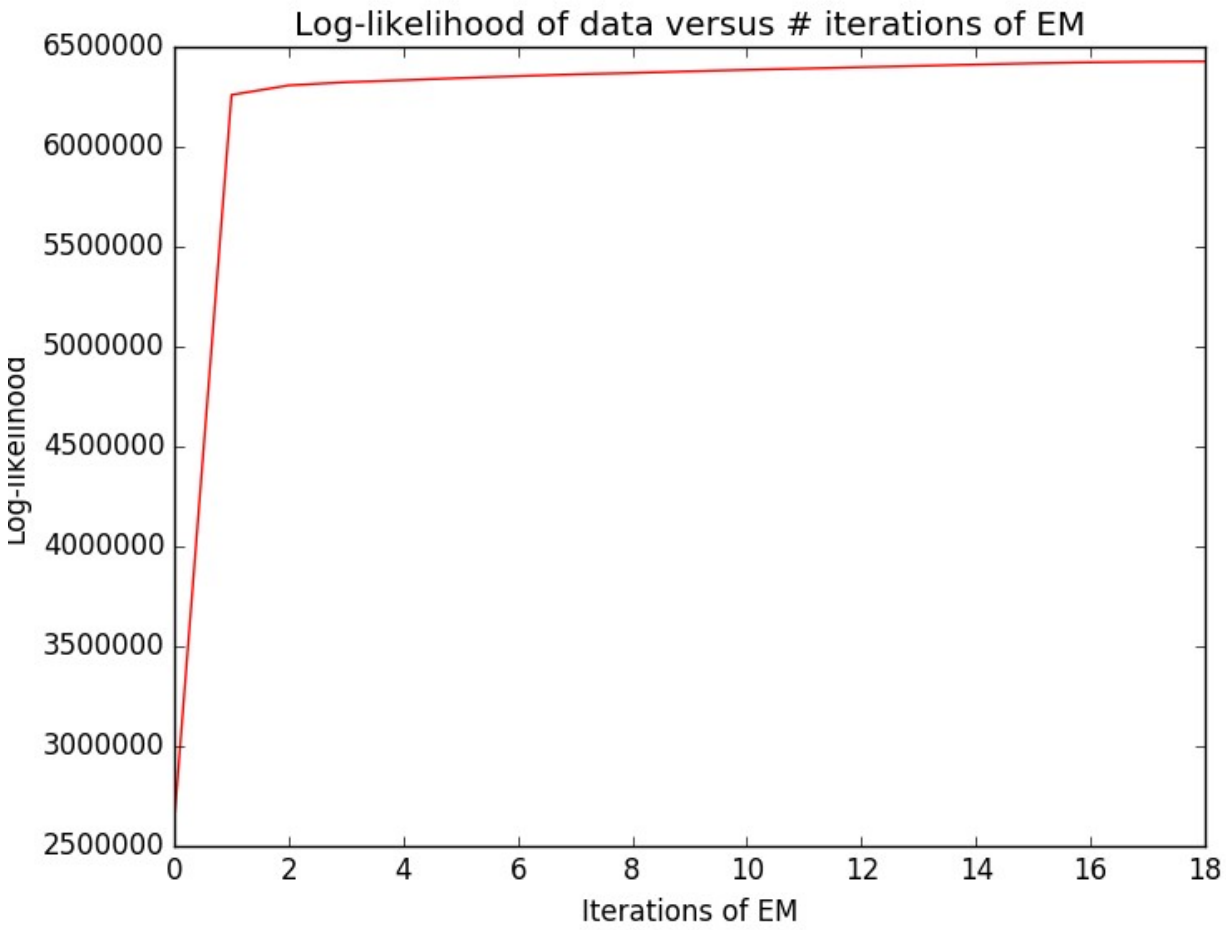
It appears to be able to converge in about half the number of iterations when initialized using kmeans (it converges in about 2 with k-means initialization, versus 4 for the random initialization).

Here are the plots and the respective log likelihood values for each initialization:



Random Initialization

Iter 1 logLikelihood 1460746.90326
Iter 2 logLikelihood 5296203.54837
Iter 3 logLikelihood 6061729.92722
Iter 4 logLikelihood 6180544.19455
Iter 5 logLikelihood 6235440.45198
Iter 6 logLikelihood 6272706.36731
Iter 7 logLikelihood 6308332.38829
Iter 8 logLikelihood 6336578.48436
Iter 9 logLikelihood 6350316.05151
Iter 10 logLikelihood 6357376.84811
Iter 11 logLikelihood 6361102.99323
Iter 12 logLikelihood 6363014.37803
Iter 13 logLikelihood 6365022.50572
Iter 14 logLikelihood 6367075.83276
Iter 15 logLikelihood 6368763.27873
Iter 16 logLikelihood 6370634.86040
Iter 17 logLikelihood 6371743.53713
Iter 18 logLikelihood 6372580.50787
Iter 19 logLikelihood 6373243.26188
Iter 20 logLikelihood 6373868.30782



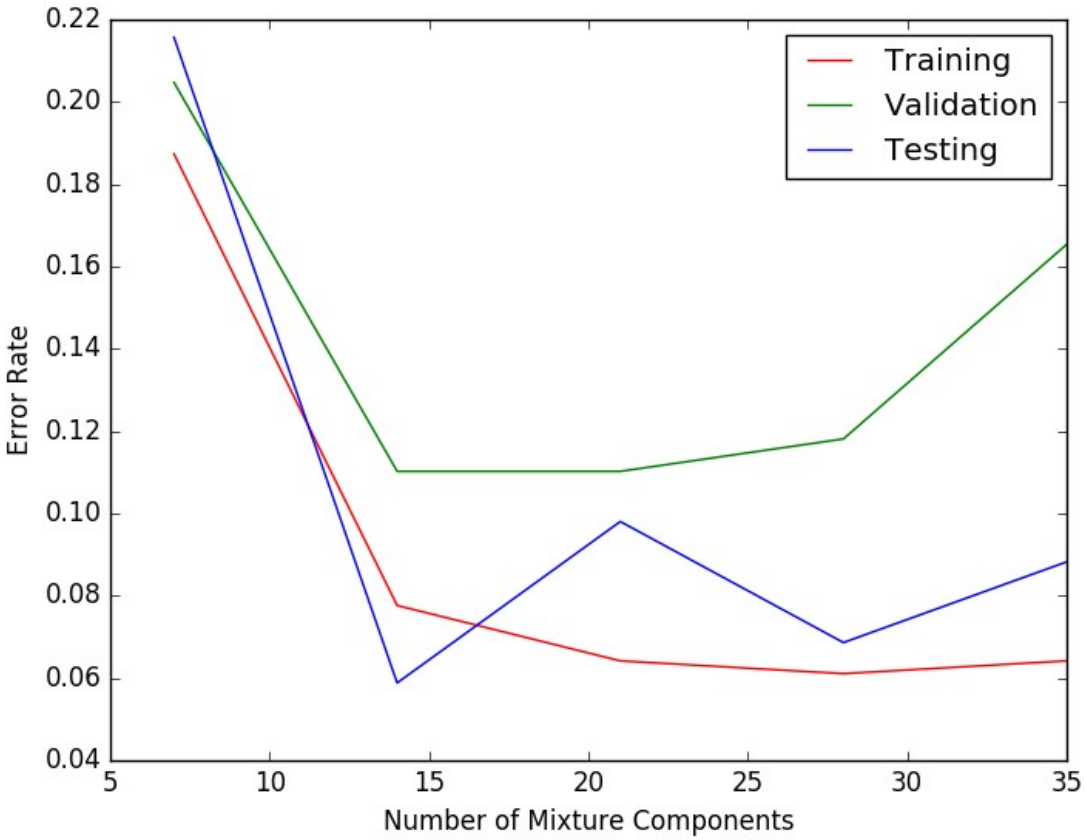
K-Means Initialization

Iter 1 logLikelihood 2636769.37076
Iter 2 logLikelihood 6316639.33040
Iter 3 logLikelihood 6357419.86528
Iter 4 logLikelihood 6368629.14155
Iter 5 logLikelihood 6372528.56291
Iter 6 logLikelihood 6373765.83019
Iter 7 logLikelihood 6374551.72336
Iter 8 logLikelihood 6375278.46797
Iter 9 logLikelihood 6375626.62920
Iter 10 logLikelihood 6376049.95730
Iter 11 logLikelihood 6376840.92613
Iter 12 logLikelihood 6378115.39571
Iter 13 logLikelihood 6379572.03243
Iter 14 logLikelihood 6381637.66661
Iter 15 logLikelihood 6384130.91941
Iter 16 logLikelihood 6385210.64565
Iter 17 logLikelihood 6385575.32116
Iter 18 logLikelihood 6385697.12528
Iter 19 logLikelihood 6385772.88903

Iter 20 logLikelihood 6385806.17281

4.4

a)



b) As the number of clusters increases, the model consequently fits the data to a greater degree. It can actually begin overfitting as k increases however. Consequently, in the limit as k increases, the error rate for the training set will decrease to nearly 0.

c) The error rate for the test set is generally higher than that of the training set. Interestingly at K = 14, the performance on the test set is actually far better than for the training set or for the validation set. In particular, the error rate for the validation and the test sets reaches a minimum at K=14. This seems to imply that the best setting of the K hyperparameter for generalization is found at K=14, with the default values of minVary and randConst provided. As K increases past 14 however, the performance worsens on the test set, seeming to imply that the model is overfitting the training data at this point. I suspect that the reasons for the large discrepancies between the training set and testing set error rates stems from the distributions of the data. The training set is likely not very representative of the test set in this case, as the features learned during training do not seem to generalize well for the other data sets except for the specific cluster values of K=14. It could also simply be that the size of the training set is too small and its features don't represent the overall dataset's very well.