Analysis, Improvement and Working

1) **Nearest Neighbor -**

Working - The nearest neighbor algorithm works by computing a matrix norm on the difference in vectors of the test and train data. This matrix norm helps us efficiently calculate an equivalent euclidean distance between two image vectors and gives us an estimate of how close a particular image vector in the test data is to another in the training data. The closest of all gives us the Nearest Neighbor for that test vector.
This can be iterated over k times to get k nearest neighbor and a maximum likelihood estimation can then be done on these k values to get the k-nearest neighbor of a particular test image. This was not implemented in code as the problem statement mentions implementing it for just 1 Nearest Neighbor. (Also comment on a post on piazza by Prof. suggests the same)

Problems Faced - Long run-times in traditional way of calculating euclidean distances between vectors. Hence, a matrix norm was used.

For the Nearest Neighbor (1 NN - as specified the assignment problem statement and on a post in piazza) classifier, the parameters that were experimented with are -

- **Training data-set sizes -** Following are some results -

Training-set size	Run-time	Accuracy
36,976	18 min	67.23%
20,000	8 min	66.59%
10,000	4 min	65.96%

Following are some samples of images that were classified correctly and incorrectly -

Correct -

ID = test/20506775873.jpg, orientation = 270

ID = test/22390308255.jpg, orientation = 180

 **ID = test/22728648085.jpg, orientation = 90**

 **ID = test/13886434273.jpg, orientation = 0**

Incorrect -

 **ID = test/2951571365.jpg , correct = 0, predicted = 180**

 **ID = test/21305225519.jpg, correct=90, predicted=270**

 **ID = test/164225933.jpg, correct = 180, predicted = 90**

 **ID = test/7941540620.jp, correct = 270, predicted = 90**

Presence of patterns -

The samples of images shown above with correct and incorrect classification, do not represent the complete set of data. But, with these samples -

Correctly classified images mostly seem to have clear color boundaries as 3 of them belong to images of a landscape

Incorrectly classified images seem to have some notion of one color being more present than any other such that on flipping the image it would be hard to make out the top and bottom of the image.

Again, this is not a representative of the entire data set, but just an intuition we got from a samples taken from classification of a smaller dataset.

2) Adaboost -

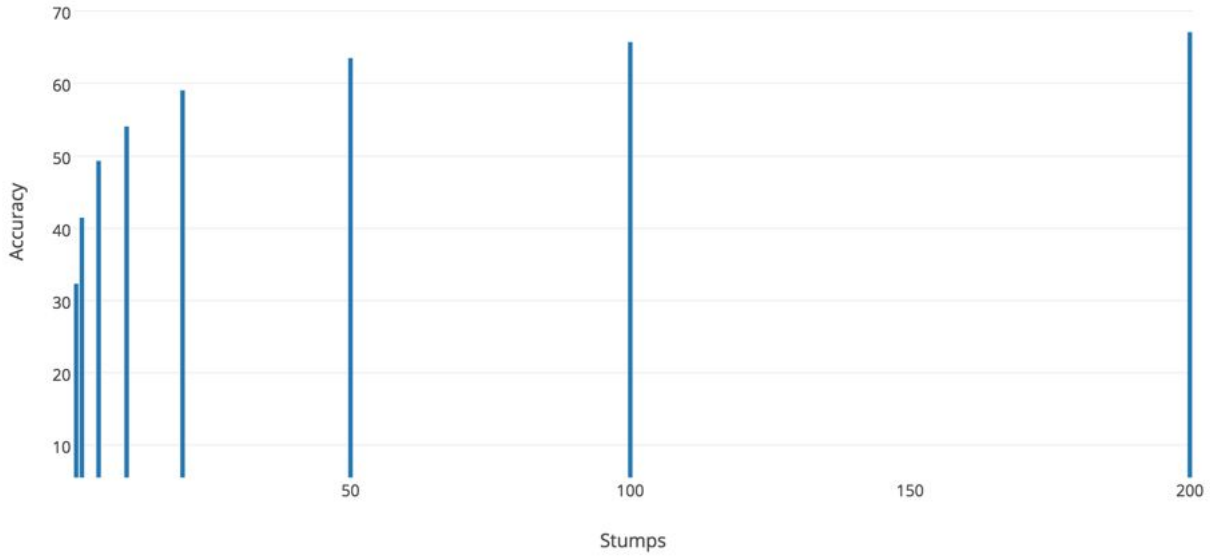
Algorithm:

1. Generate n stumps for each orientation
2. Stump generation is basically done as follows:
    a. Initial weight assigned to all training examples as 1/number of training examples
    b. Generate two distinct random index of the vector which act as classifiers
    c. Calculate error based on how accurate the classifier is
    d. Calculate alpha based on error
    e. Reassign weights of each training samples as follows:
        i. For right: 0.5/divided by number of right samples
        ii. For wrong: 0.5/divided by number of wrong samples
    f. Repeat step b to e with new weights for each classifier
3. For testing run all the stumps of each orientation and add or subtract the value of alpha for that stump based on how it classifies the sample
4. The Orientation with the highest value is assigned to the sample

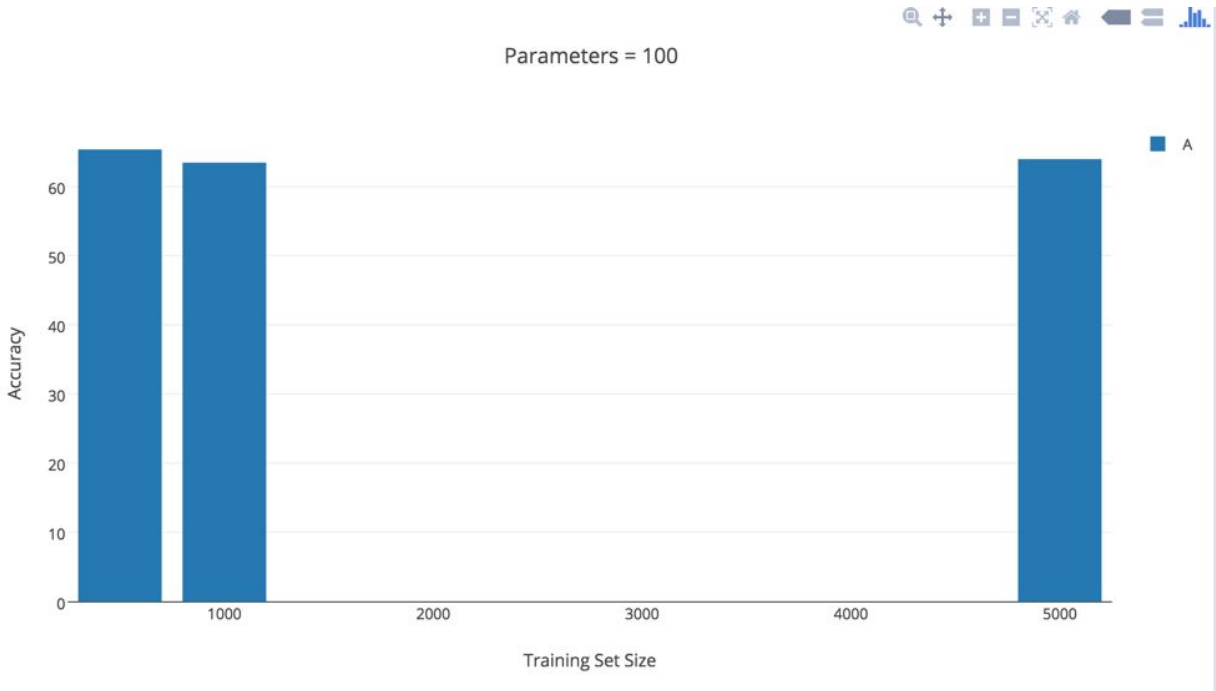
For Adaboost the only parameter to be considered is the number of stumps that constitute the classifier.
Things observed:

1. The better classifier has a higher alpha value
2. Number of stumps increases the accuracy in most of the cases
3. After 30 stumps the accuracy is generally constant with an accuracy of about 65-68%

The learning capacity of the model using random points for stumps is really low. That is why it reaches a saturation at 65-67% accuracy. We need a much more complicated model to improve the accuracy to a higher value.



Having tested with different training set sizes of 500, 1000, 5000 for each orientation the accuracy remains consistent.



Presence of patterns in correct and incorrect classifications - No specific patterns were observed in incorrect and correct classification of images. They were being classified so at random.

3) **Neural Networks:**

Below is the default configuration of our Neural Network:
Number of layers = 3 (Input Layer, Hidden Layer and the Output Layer)
Default number of neurons in hidden layer = 4
Initialization of neuron weights: Random weights between 0.02 and 0.2.
Activation Function: Sigmoid
Decreasing Learning rate = Starts from 3.6 and halves at every 5th epoch.
Number of epochs = 25
Avg. Accuracy Received = 69.4%

Algorithm Basics:

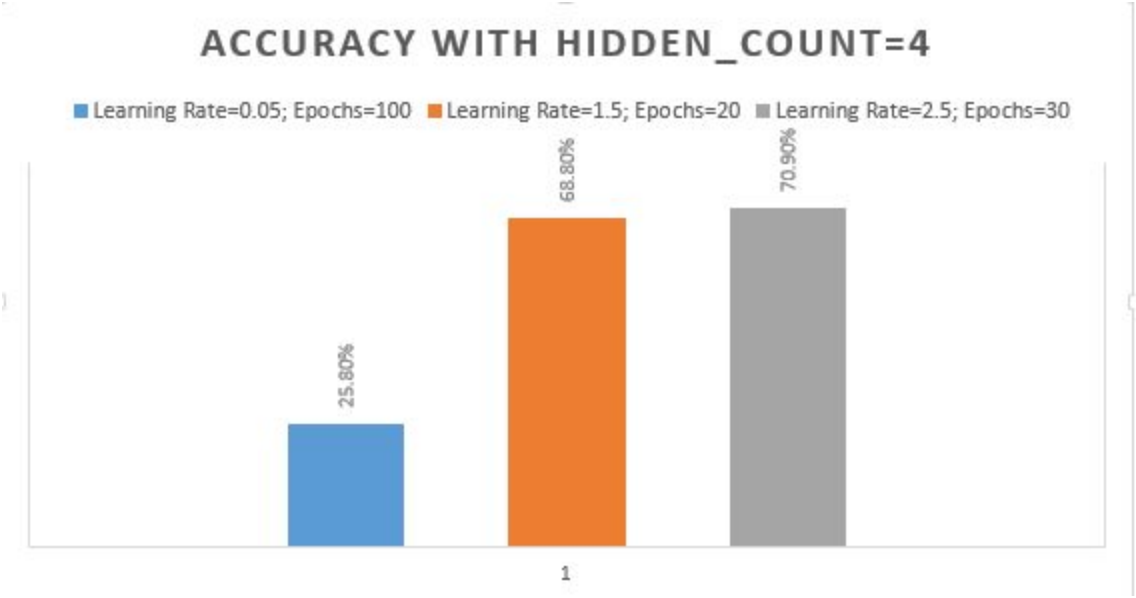
- **Neural Network initialization:** We create the hidden and output layers, and then randomly initialize the weights+bias for each neuron in both the networks.
- activateNeuron() computes the raw weight of the neuron which is then passed to the transferActivation() function which transforms the output value using Sigmoid function.
- forwardPropogate() method carry-forwards the output of one layer to the next layer.
- backPropagateError() method back-propagates the error starting from the outputLayer in reverse direction.
- updateNeuronWeights() method updates the weight of each neuron in the network wrt. the input values.

Training the Network: We have used the stochastic gradient approach wherein we complete the whole cycle of forward-propagation, back-propagation and update-weight for each training instance. For this reason, if there is a very large number of neurons in hidden layer, then training will take quite some time since we are mandatorily training the network for 25 epochs. However during experimentations we trained using 20 neurons in hidden layer and obtained a poor accuracy. We suspected overfitting and hence try to keep the hidden count in the range of 3 to 8.

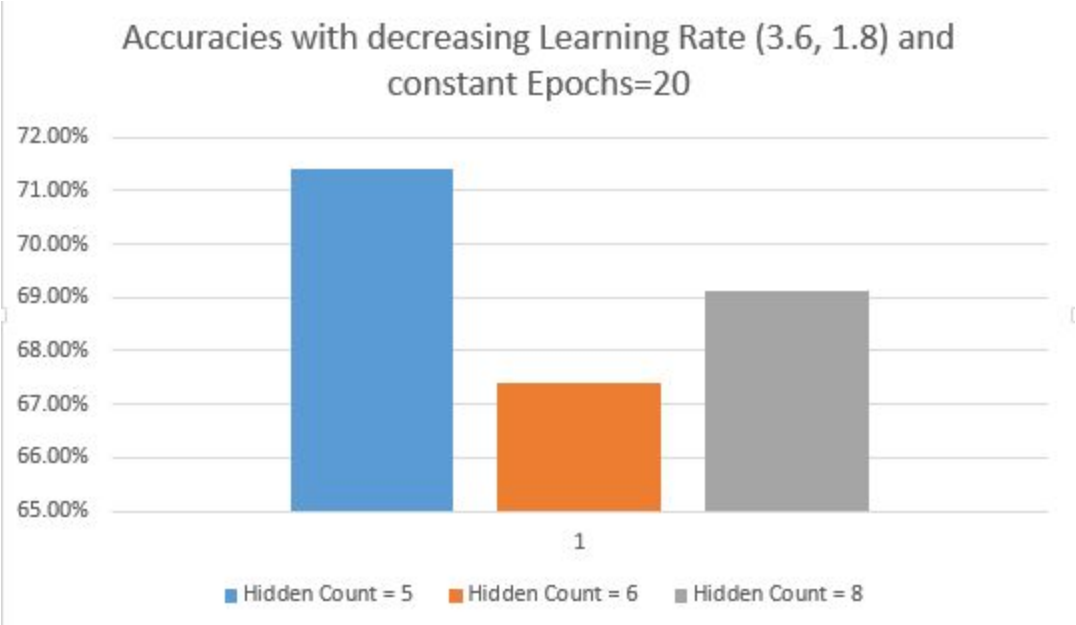
Note: While running our script in best mode, our script ignores the optional parameter - model_file as our script sets the default parameters inside itself.

Below are the graphs obtained from those experiments:

- Keeping hidden_count constant=4, we changed the learning rate and number of epochs (number of iterations on the training data)



- Next we tried to train the network with varying learning rate. Initially the learning parameter was set to a high value and after every 10 epochs it was reduced to half.



- In our next experiment, we kept the decreasing learning rate as above, epochs = 20, but increased the hidden count to 20. However with this setting we got an accuracy of 25.5%. This might be due to overfitting.
- The **highest accuracy of 72.1%** received was with the following configurations: hidden count = 10, constant learning rate = 1.8 and epochs = 25.

- In another experiment, we tried to use the hyperbolic tangent function for neuron activation. However surprisingly the accuracy dropped to 30.6%.
- Next we tried to change the way we initialized neuron weights. Earlier we assigned random weights in the range 0 to 1. In this experiment we tried to initialize with random weights in the range 0.02 to 0.2. At the same time we used decreasing learning rate after every 5 epochs and the average accuracy received is 70.8%.
- Finally we tried comparing the training time of traditional gradient descent against the stochastic gradient descent and noticed that the traditional gradient descent took nearly half time to train. At the same time, the accuracy dropped to 25.8%, so in our finally Neural Net configuration we decided to stick with the stochastic gradient descent approach for training.
- After doing the above experiments, we have decided to keep the final configuration of our neural net as follows: Epochs=25 and decreasing learning rate starting with 3.6 and halved at every 5th epoch. The activation function would be sigmoid and the initial weights of neuron will be assigned randomly within the range 0.02 to 0.2. The reason for this is that with a random range of 0 to 1, the initial squared error was about 37k, while with a random range of 0.02 to 0.2, the initial squared error is around 14k. So with small initial weights we hope to converge more quickly and accurately. Ideally we would want to have the hidden count of 4 but it comes from the command line so we don't have control over it.