Assignment #2: CIFAR-10 Image Classification using Fully Connected Neural Network

Santosh Kumar Aenugu | 933-197-448

1.

MLP. error_rate_test = ErrorRate.compute(output, target) – in line 131 in MLP.py
MLP.error_rate_train = ErrorRate.compute(output, target) – in line 137 in MLP.py

ErrorRate.compute(output, target) calls util's metrics file class **ErrorRate.compute(self,predictions,target)** which computes the errors and hence the error rate.
The result of MLP.error_rate_test is the error rate in decimals [such as 0.26] which upon multiplying with 10 and subtracting from 100 gives the accuracy as 100-26 = 74 %, hence it is used for evaluating the accuracy,

The sub-gradients of the required weights W1 and W2 are computed according to the loss's CrossEntropyLoss. crossEntropyGradient(self, output, target) function – in loss folder, cross_entropy.py line 19.
The forward pass/backward pass and gradient are separate for each layer and can be found in the Layers folder such as : [linear.py, relu.py, sigmoid.py,softmax.py]

2.

for iter, batch in enumerate(data.get_train_batches(mini_batch_size)) – in line 91 in MLP.py
performs stochastic mini-batch gradient descent training by selecting the given mini batch size of training examples for each epoch and then choosing them. We gain following advantages by using that,

a. For each epoch, data shuffle happens.
b. Mini- batches are formed
c. Get_train_batches(mini_batch_size) internally calls **_train_mini_batch(train_x_batch, train_y_batch,learning_rate, momentum, l2_penalty)** which returns error for a given mini-batch.
c. For each epoch, the training accuracy, test accuracy,test objective and training objective can be calculated easily.

The momentum approach is used as shown above to compute permuted data sequence.

3. In the hyperparameters, some of the parameters tuned for can be found:
The following parameters are tuned by me for performance analysis:

a. Learning Rate: [0.001, 0.005, 0.01, 0.05, 0.1, 0.3, 0.5, 0.7, 0.9]
Test accuracy : [0.8, 0.81, 0.8, 0.81, 0.8, 0.8, 0.8, 0.81, 0.8]
Accuracy was consistent without many deviations, I analyze this to be caused by other constants considered such as batch-size=50 which led to less deviation.

b. Hidden Units: [10, 20, 50, 100, 256, 500, 700, 1000]

Test accuracy : [0.81, 0.81, 0.82, 0.83, 0.83,
0.83,0.83, 0.83]

Accuracy change over test data was low but increasing with saturation reaching at hidden units=~256 or so indicating large no. of hidden units give us better accuracy until saturation.

c. Batch size: [10, 50, 100, 200, 500, 1000, 5000, 10000]
   Test accuracy : [0.81, 0.81, 0.8, 0.79, 0.78, 0.76,
   0.69, 0.68]
   As batch size increases, accuracy decreases. Hence considering smaller batch sizes gives better results but more time for computation in whole epoch is required. Ideally, batch size of ~100 was good for training the data set.
d. Momentum: [0.2, 0.4, 0.6, 0.7, 0.8, 0.9]
   Test accuracy: [0.79, 0.8, 0.81, 0.81, 0.81, 0.81]
   Accuracy increases with Momentum but saturates at around 0.6 or 0.7 which is ideal for training.

Analyzing all hyper-parameters, optimum values with trade-offs between computation and accuracy lead me to consider the following case:

Epochs: ~50
Learning Rate : 0.001
Momentum : ~0.7
Batch Size : ~100
Hidden Units :~ 100

which gave astonishing results of

Training Accuracy : ~99% ; Test Accuracy : ~84%

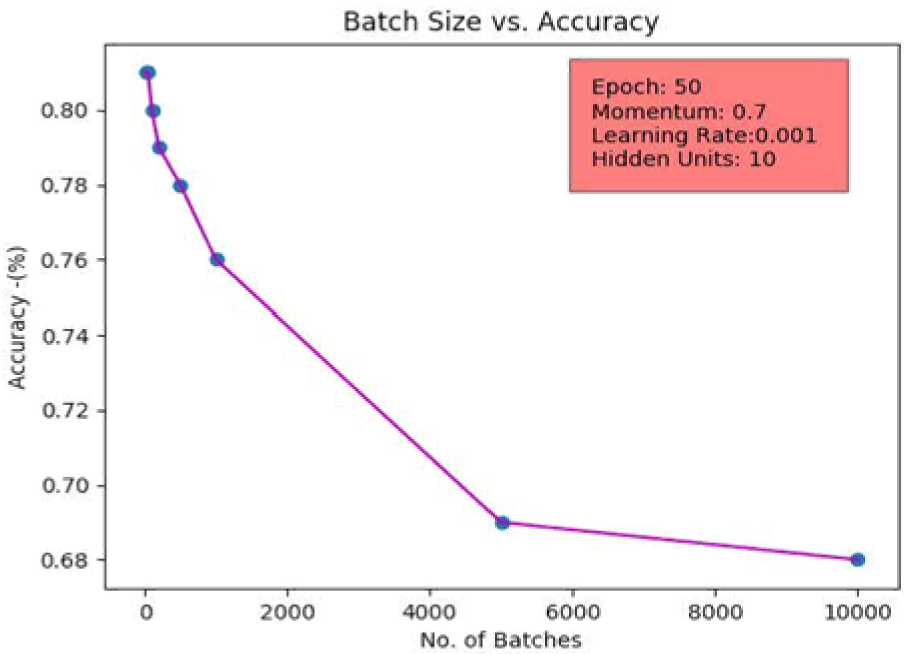
4. The following lines of code from line 127 to 145 in MLP.py evaluate the training objective, testing objective, training misclassification error rate and testing misclassification error rate as follows:

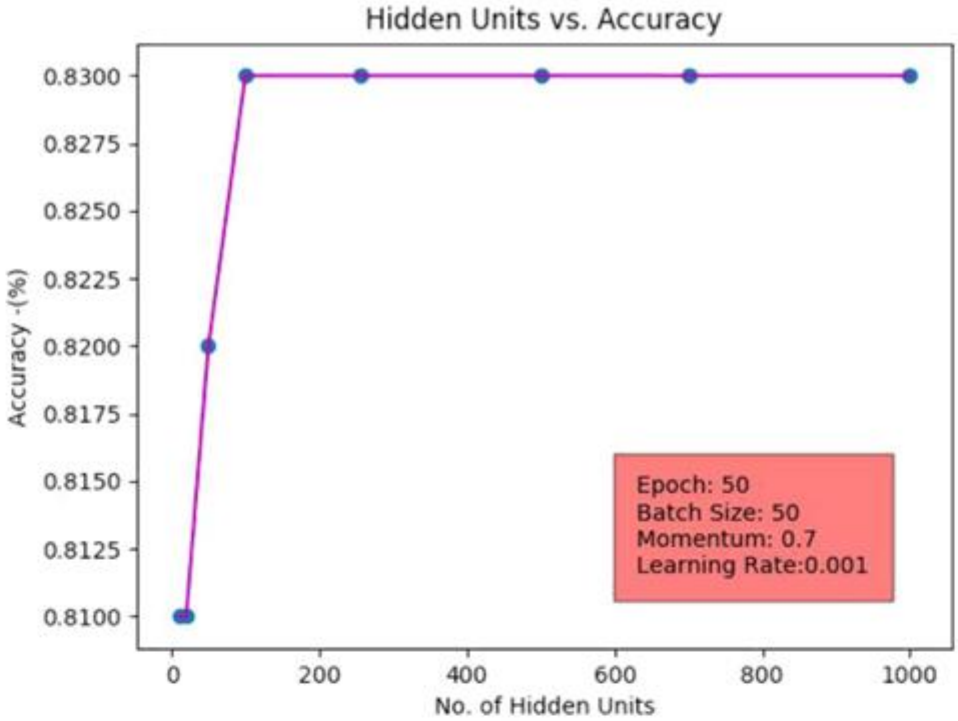
```
    target = data.get_test_labels()
x = data.get_test_data()
output = net.forward(x)
loss_avg_test = test_objective.compute(output, target)
error_rate_test = errorRate.compute(output, target)

target = data.get_train_labels()
x = data.get_train_data()
output = net.forward(x)
loss_avg_train = training_objective.compute(output, target)
error_rate_train = errorRate.compute(output, target)

elapsed = timer.getElapsed("epoch")

print("End of epoch:\ttest objective: {0}\ttrain objective:
{1}".format(loss_avg_test,loss_avg_train))
print("\t\ttest error rate: {0}\ttrain error rate:
{1}".format(error_rate_test,error_rate_train))
```

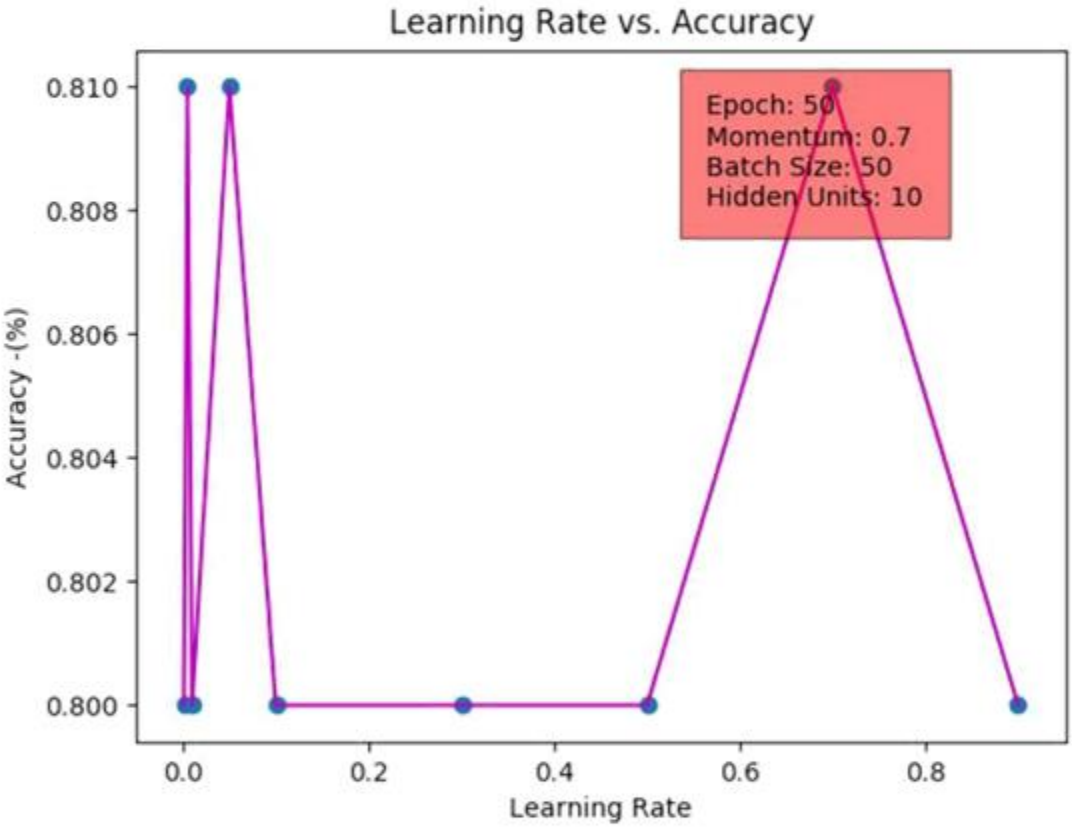
```
print("Finished epoch {1} in {0:2f}s.\n".format(elapsed, epoch))
monitor.recordEpoch(epoch, loss_avg_train, loss_avg_test,
                    error_rate_train, error_rate_test, elapsed)
```

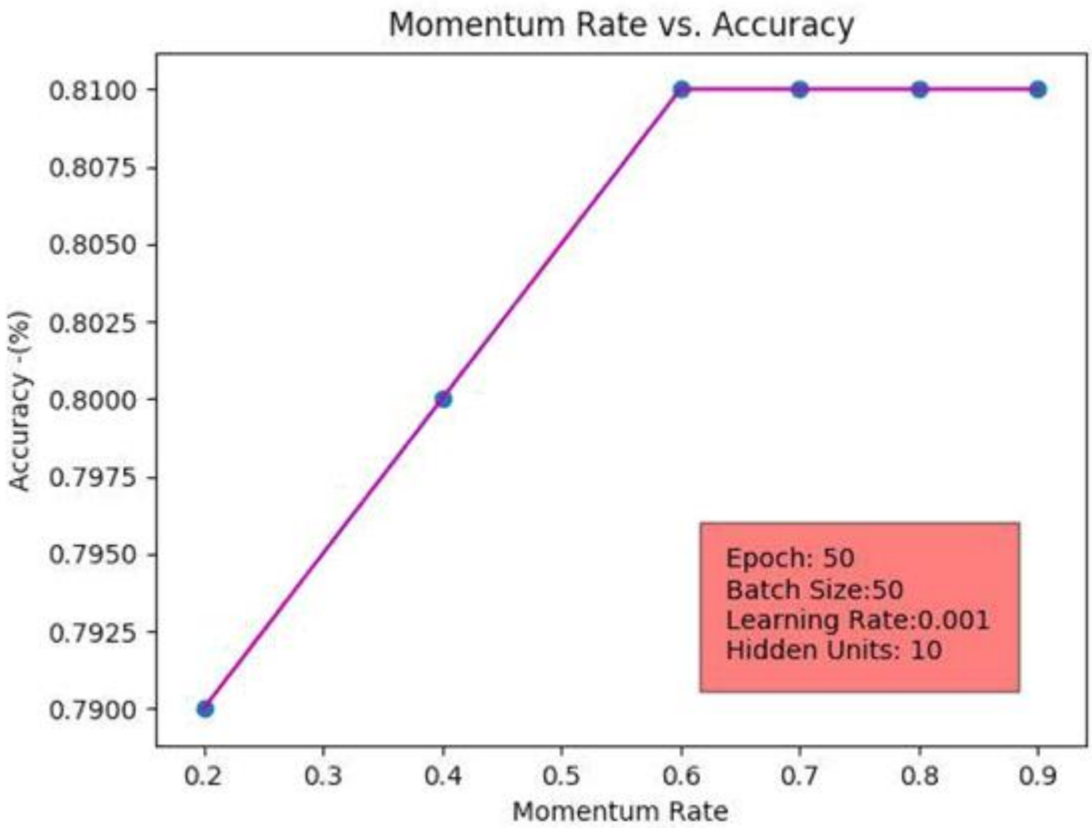
At each iteration of the epoch, the above values are printed using the print statements shown above.

5.




Hidden Units vs. Accuracy
Accuracy -(%)
0.8300
0.8275
0.8250
0.8225
0.8200
0.8175
0.8150
0.8125
0.8100
Epoch: 50
Batch Size: 50
Momentum: 0.7
Learning Rate:0.001
0
200
400
600
800
1000
No. of Hidden Units


Learning Rate vs. Accuracy
Accuracy -(%)
0.810
0.808
0.806
0.804
0.802
0.800
Epoch: 50
Momentum: 0.7
Batch Size: 50
Hidden Units: 10
0.0
0.2
0.4
0.6
0.8
Learning Rate



6. Performance:

   Present accuracy: ~83% on train set and ~ 75%+ for test set.

   Stable network to model given data gotten by tunings:

   a. Normalizing training and test data using mean and standard deviation can help avoid underflow/overflow.
   b. Remove Cross-Entropy values of 0 and 1 for neighbors substituting by 0.01 and 0.99 to avoid under/over flow. It can be avoided if data is normalized.
   c. Using xavier-initialization for weights instead of random initialization in case of Relu Units which helps to normalize Relu's output.