CS 535 Deep Learning, Assignment #2
Gulshankumar Bakle

Que 1.

The function that evaluates the trained network is written within MLP() class defined as

def evaluate(self, x, y,momentum,l2_penalty):

```
    ltransform_layer_1=self.ip_to_hl.forward(x)
    reluOp=self.relu.forward(ltransform_layer_1)     #output of layer 1
    ltransform_layer_2=self.hl_to_op.forward(reluOp)
    sigmoidOp=self.sigmoidce.forward(ltransform_layer_2)    #output for layer layer 2
    y_test=np.clip(sigmoidOp,1e-12, 1 - (1e-12))

    self.y_test_pred=y_test

    self.training_loss = self.cross_entropy_loss(y_batch, self.y_train_pred, l2_penalty)
    self.training_misclassification = self.misclassification_rate(y_batch, self.y_train_pred)

    return self.test_loss,self.test_misclassification
```

Here ip_to_hl.forward() performs the linear transformation of input, which is followed by relu output. Since its just one hidden layer, the output of relu is fed to linear transform and sigmoid transform respectively.
Functions **cross_entropy_loss and training misclassification_rate calculates the losses and error due to misclassification after training is done.**

For calculating subgradients of w1 and w2, are present in ***def train(self, x_batch, y_batch, learning_rate, momentum,l2_penalty):***

Here x_batch and y_batch are batches from training examples which are called for every batch size. Learning rate is the rate at which the network learns (alpha), and l2_penalty is a regularization term.
After forward propagation, during back propagation functions LinearTransformation.backward, Relu.backward calculates the sub gradients for w1, w2 respectively.

Que 2

Function def train() in MLP class has updates stochastic mini batch gradient descent training. For every batch, mlp.train() is called which performs forward and backward propagation followed by weight updates.

def update_wt_bias(self,old_momentum_w1,old_momentum_b1,de_by_dw,de_by_db, learning_rate,momentum,l2_penalty):

momentum_w= momentum*old_momentum_w1 - learning_rate*(de_by_dw+ l2_penalty*self.w)

momentum_b = momentum*old_momentum_b1 - learning_rate*(de_by_db + l2_penalty*self.b)

self.w = self.w+momentum_w

self.b =self.b+momentum_b

This function updates the weights and biases using momentum and l2 regularization.

Que 3

Best accuracy achieved:

On various combinations of number of epochs, batch size, learning rate, momentum, l2 regularization and number of hidden units , the best network gave was **testing accuracy with 82.25%**. This was achieved when the network is trained for 100 epochs, **learning rate as 0.0001**, **momentum as 0.8, 512 hidden layers, and l2 penalty as 0.0001.** The training accuracy with combination was around 92%, which I think has overfit the training data to some extent.

Que 4.

In this program, for training and testing objective we have used average cross entropy loss, which is defined as follows

def cross_entropy_loss(self,y,y_pred,l2_penalty):

cross_entropy = (np.sum(np.square(self.ip_to_hl.w)) + np.sum(np.square(self.hl_to_op.w))) * (l2_penalty/ (2 * len(y))) + (np.dot(np.transpose(y), np.log(y_pred + 1e-12)) + np.dot((1 - np.transpose(y)), np.log(1 - y_pred + 1e-12)))

#print(cumulative)

avg_batch_cross_entropy_loss = (-1.0)* np.sum(cross_entropy)/len(y)

return avg_batch_cross_entropy_loss

y is the actual output or label in training and testing set; y_pred is the class predicted by our model and l2_penalty is L2 regularization.

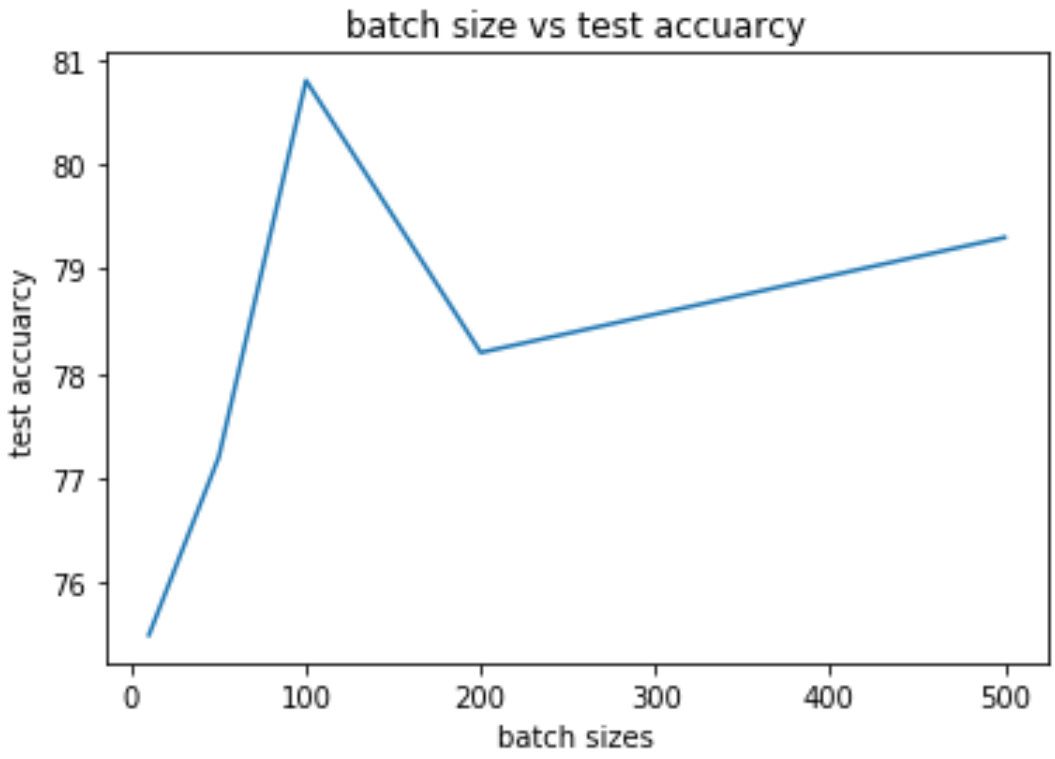
For training and testing misclassification rates, I have defined following function
def misclassification_rate(self,y,y_pred):

y_pred = np.where(y_pred >= 0.5, np.ones(y_pred.shape), np.zeros(y_pred.shape))
test = (y_pred == y)
num_of_misclassifications = len((np.where(test==False))[0]) #shortcut
#print("checkssss",misclassifications)
#time.sleep(11
return num_of_misclassifications

Here for rounding off the decimal values to either 0 or 1, np.where function is used. It calculates the misclassifications done by network.

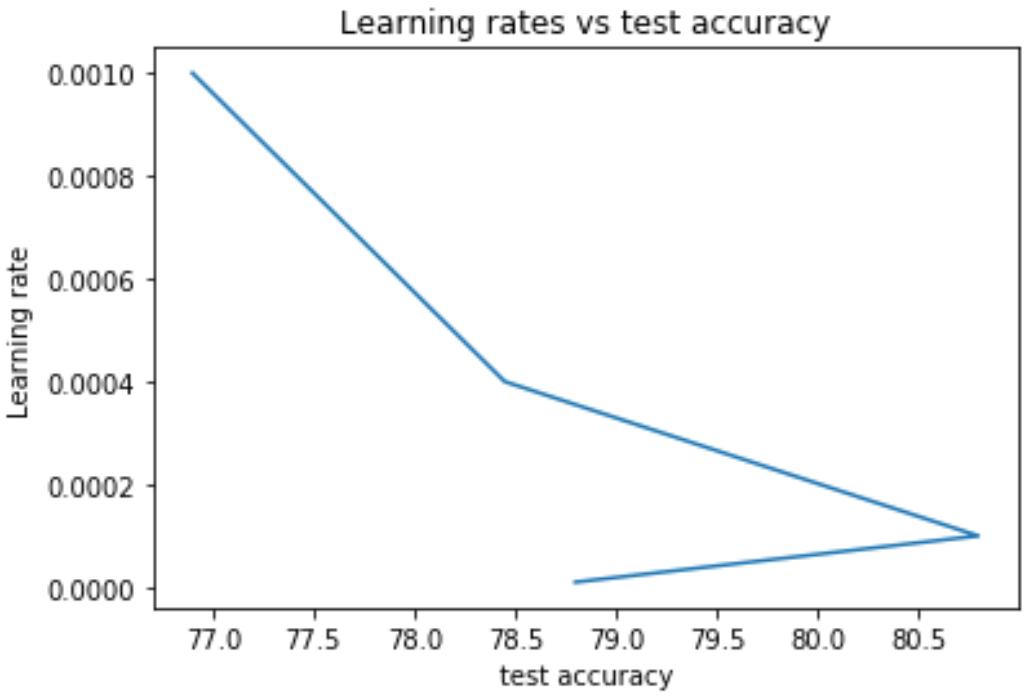
Que 5

I. test accuracy with different batch size:

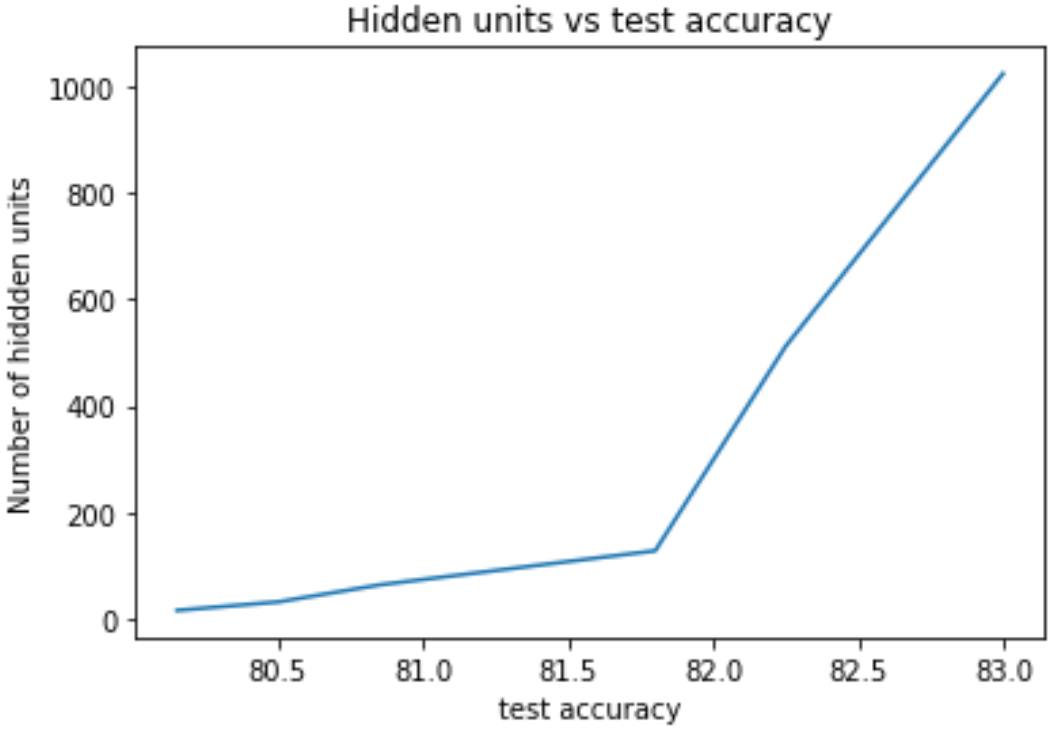


II. Learning rates vs accuracy :

This below graph is for configuration is with momentum 0.8, 32 hidden units and batch size as 100.



III. Hidden units vs accuracy:



As can be seen from above that as number of hidden units increases the accuracy increases but the training time becomes slow enough. The above

results were obtained with learning rate 0.0001, momentum 0.8 and batch size 100.

Que 6

Effect of momentum and learning rate:

On reducing the momentum to 0.7, and increasing learning rate by a factor 10, it was observed that testing accuracy drops to 73%. This means though a factor of 0.1 change in momentum does not impact our network much, but an increase in learning rate reduces the testing accuracy, indicating model reaching at sub optimal weights.

Effect of increasing the number of epochs:

Training the network for a much larger number of epochs, say 1000, doesn't improvise the accuracy of our network, as it was observed that model overfits the data. The training accuracy reaches upto 97.5% and testing accuracy dropping to 77.5%. Though training loss reduces over a larger number of epochs but test losses increases which indicates unable to predict new data correctly.

Effect of increasing or decreasing number of hidden units:

It was observed that for hidden units 16,32,64,128,256,512 and learning rate 0.0001, momentum 0.8, the testing accuracy increases as 80.15,80.5,80.85,81.8,82.25 respectively. But it was observed that though there is improvement in testing accuracy as we increase the number of neurons in the hidden layer, the training time increases along with it. So 32 to 128 hidden units could be an apt choice for this kind of problem.