1. Linear Regression

1.1 Loss function and Gradient

In this problem, the following Mean Squared Error loss function will be considered:

$$\mathrm{L} = L_D + L_W = \sum_{n=1}^{N} \frac{1}{2N} \left\| W^T \boldsymbol{x}^{(n)} + b - y^{(n)} \right\|_2^2 + \frac{\lambda}{2} \|W\|_2^2$$

The analytical solution for the gradient is derived as:

$$\frac{\partial \mathrm{L}}{\partial \mathrm{W}} = \frac{1}{\mathrm{N}} \sum_{n=1}^{N} \boldsymbol{x}^{(n)} \left(W^T x^{(n)} + b - y^{(n)} \right) + \lambda W$$

$$\frac{\partial \mathrm{L}}{\partial \mathrm{b}} = \frac{1}{\mathrm{N}} \sum_{\mathrm{n}=1}^{\mathrm{N}} \left(W^T x^{(n)} + b - y^{(n)} \right) + \frac{\lambda}{2} \|W\|_2^2$$

The snipped of code is attached at the end of the report.

1.2 Gradient Descent Implementation

The Gradient Descent algorithm used to update the weights and bias term is as follows:

$$\mathrm{W} = \mathrm{W} - \alpha \frac{\partial \mathrm{L}}{\partial \mathrm{W}}$$

$$\mathrm{b} = \mathrm{b} - \alpha \frac{\partial \mathrm{L}}{\partial \mathrm{b}}$$

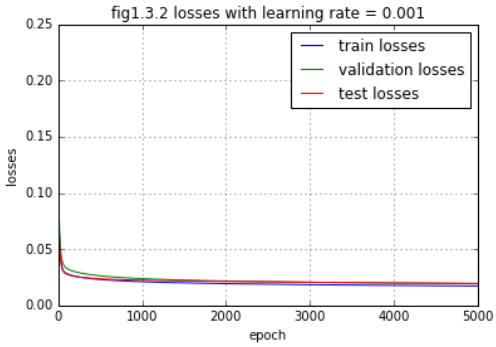
for alpha being the learning rate.

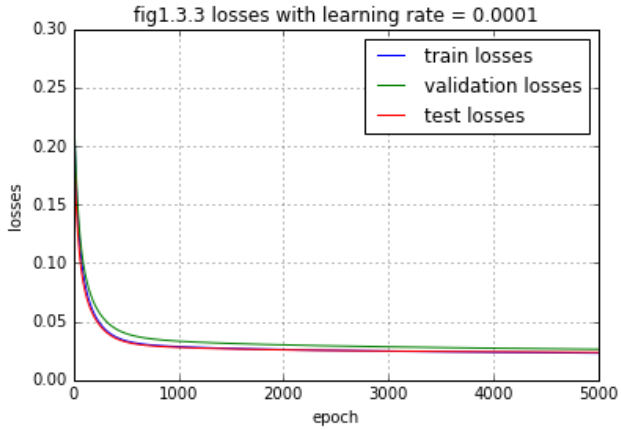
1.3 Tuning the learning rate

Test your implementation of Gradient Descent with 5000 epochs and λ = 0. Investigatethe impact of learning rate, α = 0.005; 0.001; 0.0001 on the performance of your classifier. Plot the training, validation and test losses. What is the impact of modifying α against the training time? Final classification

accuracy?

As shown in the plot below, the convergence is faster when the learning rate is higher. However, higher learning rate leads to larger differences in terms of final accuracy between test and training data sets. Therefore, based on fig1.3.1 and fig1.3.3, we can tell that slower learning rate leads to higher accuracy.



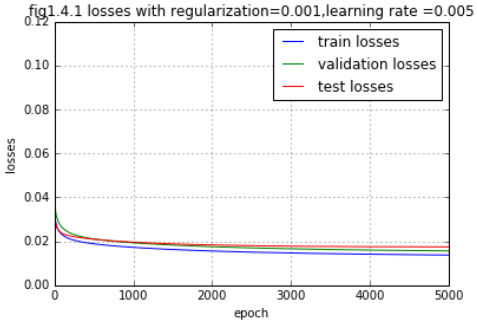


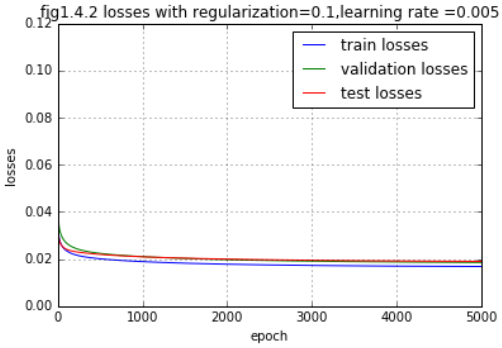
1.4 Generalization

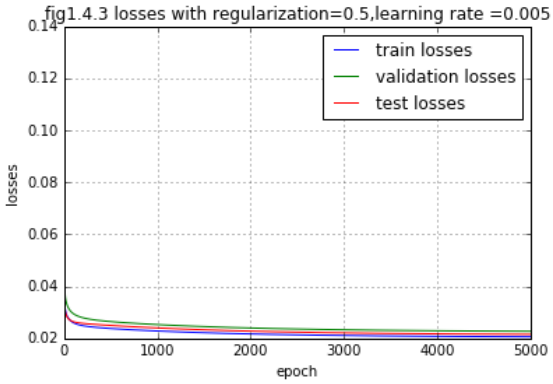
Investigate impact by modifying the regularization parameter, λ =0.001,0.1,0.5. Plot the training, validation and test loss for α = *0.*005 and report the final training, validation and test performance of your classifier. Comment on the effect of regularization on performance as well as the rationale behind tuning λ using the validation set.

As shown in the plot below, the losses for regularization = 0.5, 0.1 is much larger than the losses for regularization = 0.001. This concludes that the larger the

regularization parameter, the less accurate the model will predict, because the model relies heavier on the weights. However, the right choice of the regularization parameter will avoid the risk of overfitting. To choose the suitable regularization parameter, we can tune it with validation dataset, as it will give a more general result because we are not fitting the data using training set.



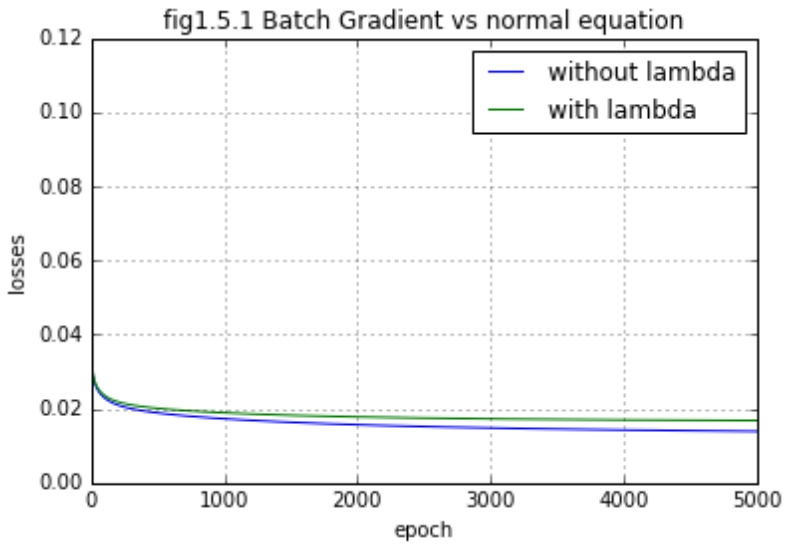




1.5 Compare batch GD with normal equation

For linear regression, you can find the optimum weights using the closed form equation for the derivative of the means square error (normal equation). For zero weight decay, Write a Numpy script to find the optimal linear regression weights on the *two-class notMNIST* dataset using the "normal equation" of the least squares formula. Compare in terms of final training MSE, accuracy and computation time between Batch GD and normal equation

As shown in fig 1.5.1 below, the batch GD with regularization parameter converges slower than the normal equation without lambda. The normal equation leads to smaller final MSE.



2.Logistic Regression

2.1 Loss Function and Gradient

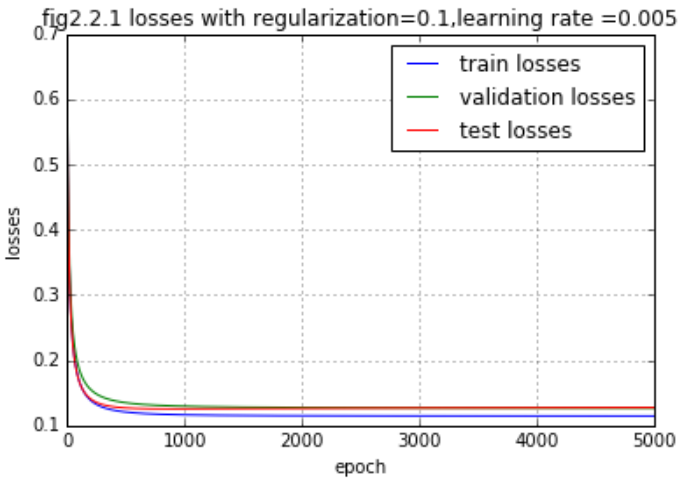
$$\frac{\partial L}{\partial W} = \sum_{n=1}^{N} \frac{1}{N}\left(\frac{-y^{(n)}x^{(n)}}{1+e^{W^T x^{(n)}+b}} + \frac{\left(1-y^{(n)}\right)x^{(n)}}{e^{-W^T x^{(n)}-b}+1}\right) + \lambda W$$

$$\frac{\partial \mathrm{L}}{\partial \mathrm{W}} = \sum_{n=1}^{N} \frac{1}{N}\left(-\frac{y^{(n)}}{e^{b+W^T x^{(n)}}+1} + \frac{(1-y)}{(1+e^{-b-W^T x^{(n)}})}\right)$$

A snipped of code is include in the appendix.

2.2 Learning

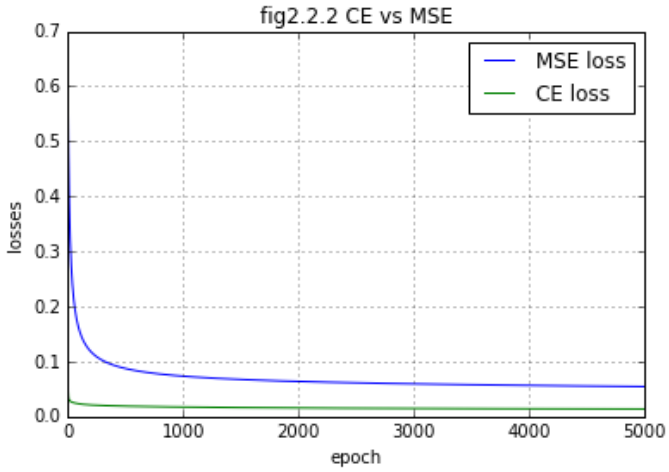
Modify the function from Part 1.2 to include a flag, specifying the type of loss/gradient to use in the classifier. Modify your function to update weights and biases using the Binary Cross Entropy loss and report on the performance of the Logistic Regression model by setting λ = 0.1 and 5000 epochs. Plot the loss and accuracy curves.



2.3 Comparison to Linear Regression

For zero weight decay, learning rate of 0.005 and 5000 epochs , plot the training cross entropy loss and MSE loss for logistic regression and linear regression respectively. Comment on the effect of cross-entropy loss convergence behaviour.

As shown in the fig2.2.2, MSE losses are much higher than the CE losses for the entire training period. Therefore, using CE for logistic regression for classification problems gives better performance. The main rationale behind this is that square loss model needs the targets to hit specific values rather than having values correspond to higher probabilities. It is really hard for the model to learn. On the other hand, cross-entropy is a better measure than MSE for classification problem because the decision boundary in classification is large.



3. Batch Gradient Descent vs SGD and Adam

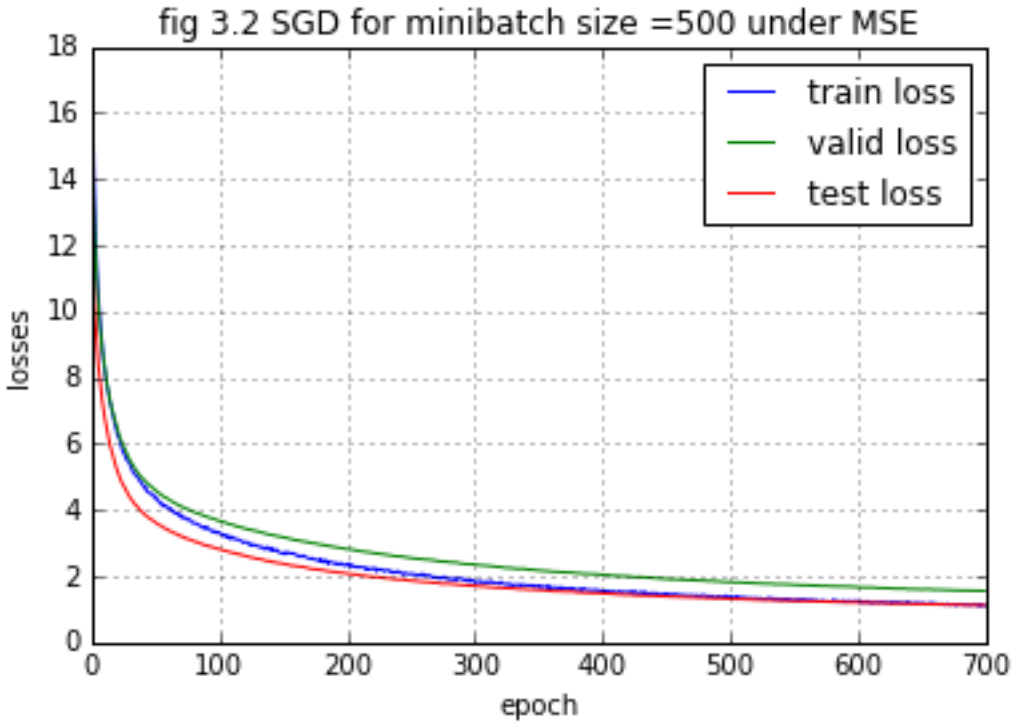
3.1 Building the computational graph

Please see a snippet of code in Appendix 3.1

3.2 Implementation of SGD

Implement the SGD algorithm for a minibatch size of 500 optimizing over 700 epochs [2], minimizing the MSE. Calculate the total number of batches required by dividing the number of training instances by the minibatch size. After each epoch you will need to reshuffle the training data and start sampling from the beginning again. Initially, set $\lambda = 0$ and continue to use the same α value (i.e. 0.001)

Total number of batches required is 7 for each epoch.

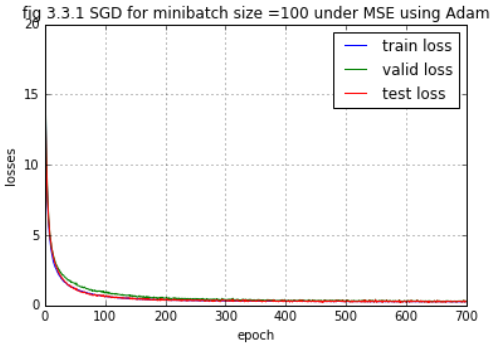


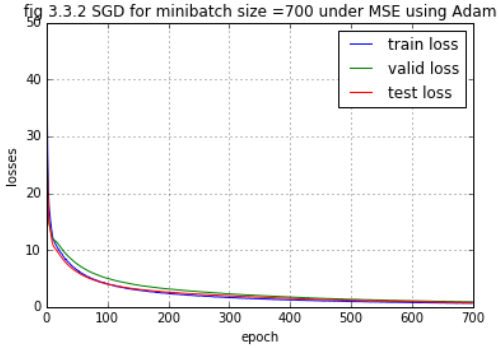
3.3 Batch Size investigation

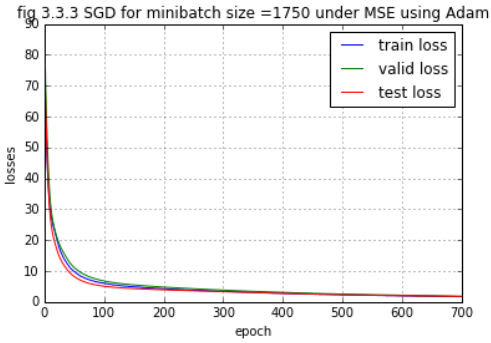
Study the effects of batch size on behaviour of the SGD algorithm optimized using Adam by optimizing the model using batch sizes of $B = 100; 700; 1750$. For each case, plot the loss and accuracy curves. What is the impact of batch size on the final classification accuracy for each of the 3 cases? What is the rationale for this?

As shown in the three plots below, the batch size has a significant effect on convergence behavior. The convergence with small batch sizes is faster compared with that of larger batch sizes. However, the smaller batch sizes also lead to some noises as shown in fig3.3.1 at the first 100 iterations. Although the final classification losses might be the same, while for a given accuracy, the larger batch

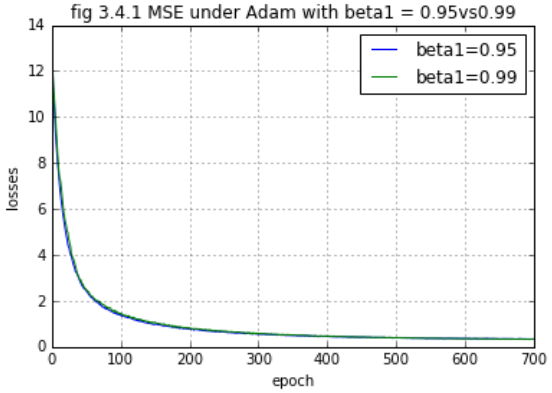
sizes will take longer computational time.

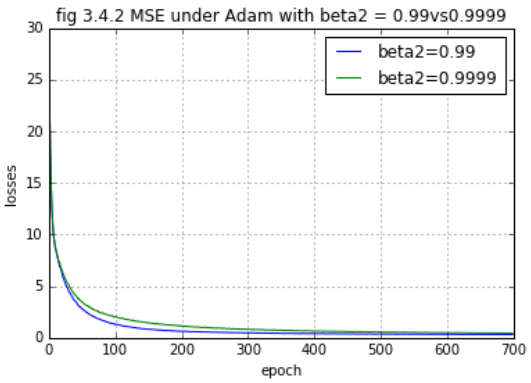


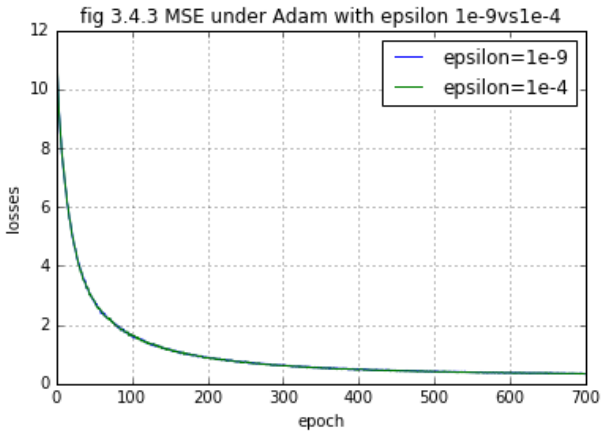




3.4 Hyperparameter investigation





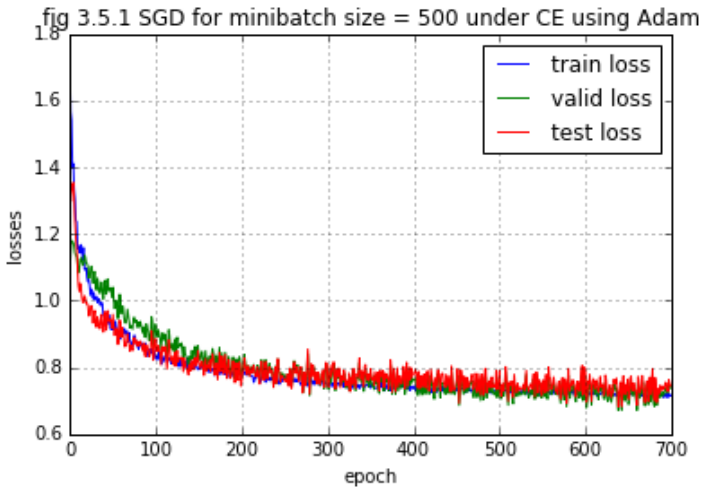


3.5 Cross Entropy Investigation

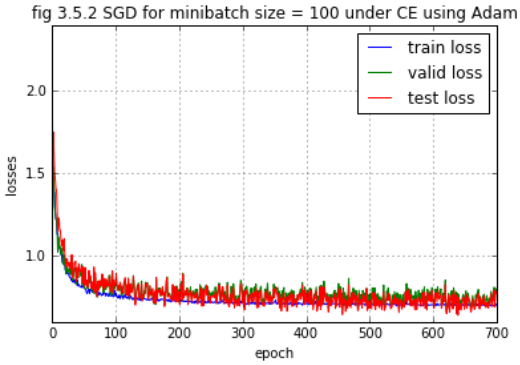
Repeat parts 3.1.2 to 3.1.4 by minimizing the binary cross entropy loss. How do the two models compare against each other in terms model performance (i.e. final classification accuracy)?

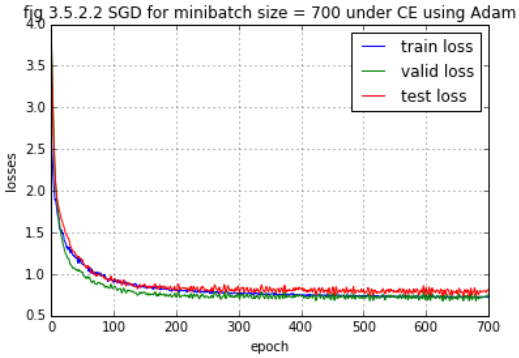
The losses from cross entropy model are smaller than the losses using MSE model. This concludes that cross-entropy is preferred for classification problem. From the probability distribution point of view, the assumption for the error in MSE follows normal distribution while the assumption for the error in CE follows binomial distribution, which implies that the CE models are doing classification. However, when using MSE models, it is basically doing linear regression which are not very accurate in classification problems.

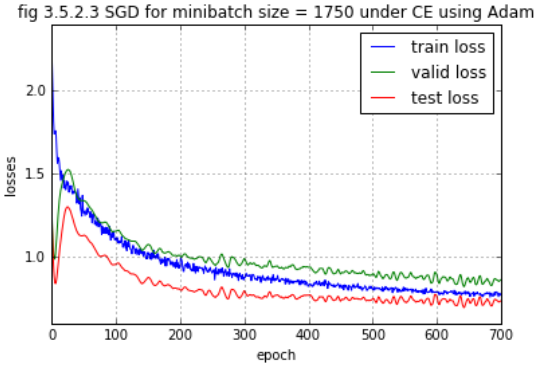
3.5.1 Implementation of SGD on CE with lambda = 0, alpha = 0.001, batch size = 500:



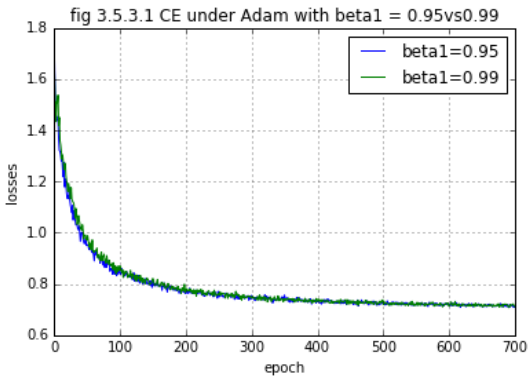
3.5.2 Investigation on batch size on CE

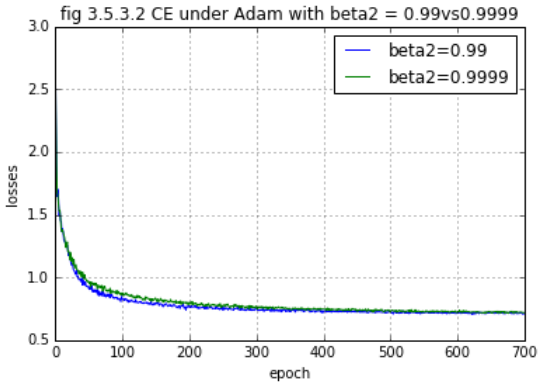


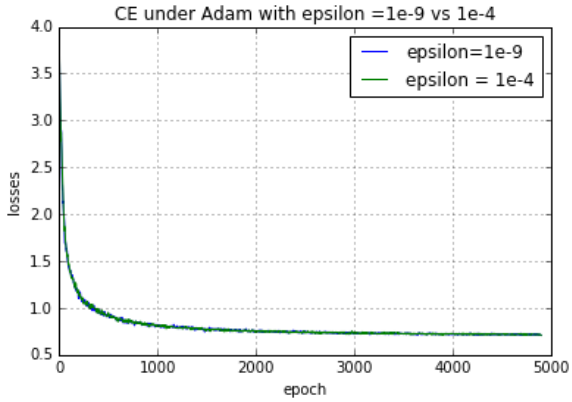




3.5.3 Hyperparameter investigation







6. Comparison against Batch GD

Comment on the overall performance of the SGD algorithm with Adam vs. the batch gradient descent algorithm you implemented earlier. Additionally, comment on the plots of the losses and accuracies of the SGD vs. batch gradient descent implementation. What do you notice about the curves? Why is this happening?

For this assignment, the batch gradient descent algorithm implemented in section 1 and 2 results higher accuracies and smaller losses than SGD algorithm with Adam shown in section 3.

Appendix
Snippet of Code

1.1 Loss Function and gradient

```
def MSE(W, b, x, y, reg):
    weight_decay = (reg/2)*np.sum(np.transpose(W)**2)
    error = (np.dot(np.transpose(W),x)+b-y)**2
    return (1/(2*len(x[1])))*np.sum(error)+weight_decay

def gradMSE(W, b, x, y, reg):
    weight_decay = reg*W
    error = (np.dot(np.transpose(W),x)+b-y)
    gradient_W = np.dot(x,np.transpose(error))/len(x[1])+weight_decay
    gradient_b = np.sum(error)/len(x[1])+(reg/2)*np.sum(np.transpose(W)**2)
    return gradient_W, gradient_b
```

1.2 Gradient Descent Implementation and 2.2

```
def grad_descent(W, b, x, y, alpha, iterations, reg, EPS,losstype="None"):
    new_W = W
    new_b =b
    loss_train = np.ones(5000)
    loss_valid = np.ones(5000)
    loss_test = np.ones(5000)
    epoch = np.arange(5000)
    if losstype == "gradMSE":
        grad_loss = gradMSE
        loss_func= MSE
    else:
        grad_loss = gradCE
        loss_func= crossEntropyLoss

    for i in range(iterations):
        gradient_W,gradient_b = grad_loss(new_W,new_b,x,y,reg)
        new_W = W - (alpha*gradient_W)
        new_b = b -(alpha*gradient_b)
        loss_train[i] = (loss_func(new_W,new_b,x,y,reg))
        loss_valid[i] = (loss_func(new_W,new_b,validData,validTarget,reg))
        loss_test[i] = (loss_func(new_W,new_b,testData,testTarget,reg))
```

```
            if np.sum(abs(new_W-W))<EPS:
                print("converged")
                break
            W = new_W
            b = new_b
        return new_W,new_b,epoch,loss_train,loss_valid,loss_test
```

2.1 Loss function and Gradient

```
    def crossEntropyLoss(W, b, x, y, reg):
            weight_decay = (reg/2)*np.sum(np.transpose(W)**2)
            y_hat = sigmoid(np.dot(np.transpose(W),x)+b)
            error = np.sum((-1)*y*np.log(y_hat)-(1-y)*np.log(1-y_hat))/len(x[1])
            return error+weight_decay

    def gradCE(W, b, x, y, reg):
            weight_decay = reg*W
            gradCE_W=(np.dot(x,np.transpose((-
1*y)*(1/(1+np.exp(np.dot(np.transpose(W),trainData)+b)))))
        +np.dot(x,np.transpose((1-y)*(1/(1+np.exp(-
1*np.dot(np.transpose(W),trainData)-b))))))/len(x[1])+weight_decay
            gradCE_b   =   np.sum((-1*y/(1+np.exp(np.dot(np.transpose(W),x)+b)))+(1-
y)/(1+np.exp(-1*(np.dot(np.transpose(W),x)+b))))/len(x[1])
            return gradCE_W, gradCE_b
```

3.1 buildGraph()

```
    def buildGraph(loss=None):
            tf.set_random_seed(421)

W=tf.Variable(tf.transpose(tf.truncated_normal((784,1),stddev=0.5)),dtype=tf.float32)
            b=tf.Variable(1.,dtype=tf.float32)
            x=tf.placeholder(tf.float32,shape=(784,None))
            y=tf.placeholder(tf.float32,shape=(1,None))
            lam=tf.placeholder(tf.float32)
            alpha= tf.placeholder(tf.float32)
            y_pred=tf.matmul(W,x) + b

            if loss == "MSE":
                        #error                                                    =
                tf.reduce_mean(tf.losses.mean_squared_error(labels=y,predictions=y_pr
                ed))
```

```
            error = tf.reduce_sum(tf.pow(y_pred, 2))/(2*3500)
            weight_decay = tf.multiply(lam / 2, tf.reduce_sum(tf.square(W)))
            total_loss = error + weight_decay

        elif loss == "CE":
            error = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels=y, logits=y_pred))
            weight_decay = tf.multiply(lam / 2, tf.reduce_sum(tf.square(W)))
            total_loss = error + weight_decay

        optimizer = tf.train.GradientDescentOptimizer(alpha).minimize(total_loss)

        return x,y,W,b,y_pred,y,total_loss,optimizer,lam,alpha
```