**ECE421 – Introduction to Machine Learning – Assignment 1**

|  |  |
| --- | --- |
| FIRST NAME | Saud |
| LAST NAME | Badar |
| STUDENT # | 1002554595 |

|  |  |
| --- | --- |
| FIRST NAME | Kyu Bum |
| LAST NAME | Kim |
| STUDENT # | 1003969100 |

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

import os

import time

from google.colab import drive

drive.mount('/content/drive')

os.chdir('/content/drive/My Drive/ECE421/Lab1')

def loadData():

with np.load('notMNIST.npz') as data :

Data, Target = data ['images'], data['labels']

posClass = 2

negClass = 9

dataIndx = (Target==posClass) + (Target==negClass)

Data = Data[dataIndx]/255.

Target = Target[dataIndx].reshape(-1, 1)

Target[Target==posClass] = 1

Target[Target==negClass] = 0

np.random.seed(421)

randIndx = np.arange(len(Data))

np.random.shuffle(randIndx)

Data, Target = Data[randIndx], Target[randIndx]

trainData, trainTarget = Data[:3500], Target[:3500]

validData, validTarget = Data[3500:3600], Target[3500:3600]

testData, testTarget = Data[3600:], Target[3600:]

return trainData, validData, testData, trainTarget, validTarget, testTarget

# Section 1: Linear Regression [18 points]

## 1) Loss Function and Gradient [4pt]

Implement two vectorized Numpy functions to compute the loss and gradient respectively. Both functions should accept 5 arguments - the weight vector, the bias, the data matrix, the labels, and λ, the regularization parameter. The loss function returns a number (indicating total loss). The gradient function should be an analytical expression of the loss function and return the gradient with respect to the weights and the gradient with respect to the bias. Both function headers are below. Include both the analytical expression for the gradient and the Python code snippet in your report.

### Answer:

"""

calculates the mean squared error and returns the total loss

W is a weight, 1D-array of size: 784 (28 x 28)

b is the bias

x is the batch of Mnist data images : (3500, 28, 28)

y is the batch of labels : (3500, 1)

reg is λ, the regularization parameter

"""

def MSE(W, b, x, y, reg):

#reshaping the batch to satisfy the size of weights for multiplication

#(x shape = (3500, 784), W shape = (784, 1))

x\_flat = np.reshape(x, (x.shape[0], x.shape[1] \* x.shape[2]))

#calculate error using the formula (e shape = (3500, 1))

e = np.subtract(np.matmul(x\_flat, W) + b, y)

#calculating overall mean squared error (loss function)

mse = np.square(e).mean()

#calculating weight decay loss

weight\_loss = reg / 2 \* (np.linalg.norm(W) \*\* 2)

#return the sum of the two losses for overall loss

return weight\_loss + mse

def gradMSE(W, b, x, y, reg):

# Your implementation here

# Reshaping part of the code

newW = np.reshape(W, (x.shape[1] \* x.shape[2], 1)) # New shape of W is [784, 1]

newX = np.reshape(x, (x.shape[0], x.shape[1] \* x.shape[2])) # New shape of x is [3500, 784]

#calculate error using the formula (e shape = (3500, 1))

e\_in = np.subtract(np.matmul(newX, newW) + b, y)

#Find the gradient of weights

w\_gradient = (2 \* np.matmul(newX.transpose(), e\_in))/(x.shape[0])

#Add the regulization factor

w\_gradient = w\_gradient + reg\*newW

#Find the gradient of the bias

b\_gradient = np.sum(e\_in) / (x.shape[0])

return w\_gradient, b\_gradient

## 2) Gradient Descent Implementation [6 pts]:

Using the gradient computed from Part A, implement the batch Gradient Descent algorithm to classify the two classes in the notMNIST dataset. The function should accept 8 arguments: the weight vector, the bias, the data matrix, the labels, the learning rate, the number of epochs 1 , λ and an error tolerance (set to 1 × 10 −7 ). The error tolerance will be used to compute the difference between the old and updated weights per iteration. The function should return the optimized weight vector and bias.

### Answer:

"""

calculates optimal values of the vectors weights and bias determined by

the gradient of mean squared error loss function (MSE) and returns these

optimal values

returns weight, then bias

W is the weight

b is the bias

x is the data

y is the labels

alpha is learning rate (step)

epochs are iterations for each batch

reg is λ, the regularization parameter

error\_tol is the error tolerance (set to 1 × 10 −7 )

lossType is the type of loss function (MSE or CE)

"""

def grad\_descent(W, b, x, y, alpha, epochs, reg, error\_tol, lossType):

trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()

train\_loss = []

test\_loss = []

valid\_loss = []

train\_acc = []

valid\_acc = []

test\_acc = []

for i in range(epochs):

train\_acc.append(accuracy(W, b, x, y, lossType))

valid\_acc.append(accuracy(W, b, validData, validTarget, lossType))

test\_acc.append(accuracy(W, b, testData, testTarget, lossType))

#find loss and gradient weight and bias determined by input of this function

if lossType == 'MSE':

loss = MSE(W, b, x, y, reg)

train\_loss.append(loss)

valid\_loss.append(MSE(W, b, validData, validTarget, reg))

test\_loss.append(MSE(W, b, testData, testTarget, reg))

weights, bias = gradMSE(W, b, x, y, reg)

elif lossType == 'CE':

loss = crossEntropyLoss(W, b, x, y, reg)

train\_loss.append(loss)

valid\_loss.append(crossEntropyLoss(W, b, validData, validTarget, reg))

test\_loss.append(crossEntropyLoss(W, b, testData, testTarget, reg))

weights, bias = gradCE(W, b, x, y, reg)

#if error is smaller than the tolerance, we have the optimal values

#for weights and bias, thus we can stop iterating

if loss <= error\_tol:

break

#update weight and bias from calculated gradient

W -= alpha \* weights

b -= alpha \* bias

return W, b, train\_loss, valid\_loss, test\_loss, train\_acc, valid\_acc, test\_acc

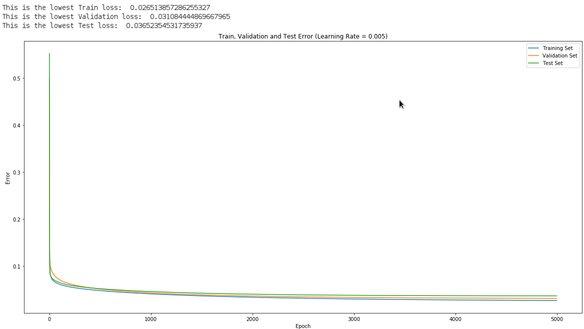
## 3) Tuning the Learning Rate [3 pts]:

Test your implementation of Gradient Descent with 5000 epochs and λ = 0. Investigate the impact of learning rate, α = 0.005, 0.001, 0.0001 on the performance of your classifier. Plot the training, validation and test losses.

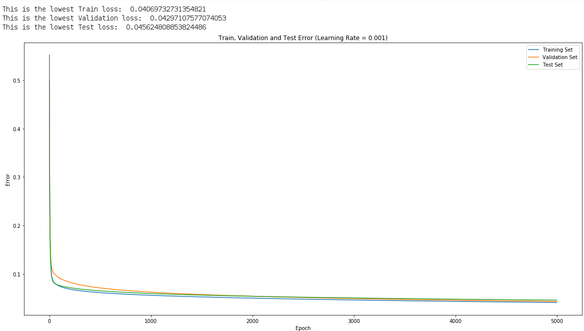
**Answer:**

**Learning Rate = 0.005 Graph**

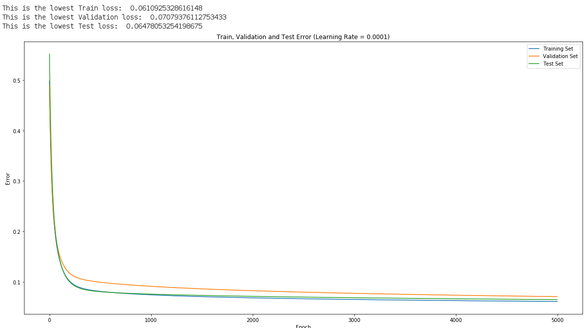
Loss/Error:

****

**Learning Rate = 0.001 Graph**

****

**Learning Rate = 0.0001 Graph**



From what we can tell all the the training, validation and the test loss are converging slowly towards zero. So the higher our learning rate is the faster the graphs converges to zero. We can tell this from the lowest losses in each graph, **0.005 is 0.0265, 0.001 is 0.0407, 0.0001 is 0.0611.** This shows the higher the learning rate is faster it converges to zero. We have to keep in mind not to increase the learning curve too much as that will cause our graph to never converge.

## 4) Generalization [3 pts]:

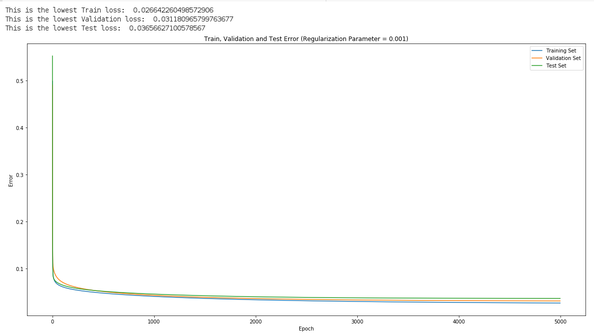
Investigate impact by modifying the regularization parameter, λ = {0.001, 0.1, 0.5}. Plot the training, validation and test loss for α = 0.005 and report the final training, validation and test accuracy of your classifier.

### Answer:

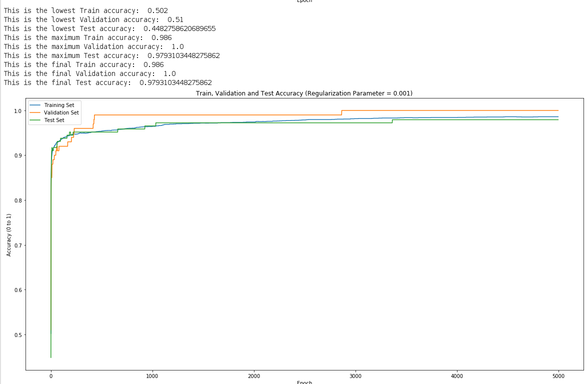
|  |  |  |  |
| --- | --- | --- | --- |
| **Regularization Parameter** | **Final Training Accuracy** | **Final Validation Accuracy** | **Final Testing Accuracy** |
| **0.001** | **98.6%** | **100%** | **97.93%** |
| **0.1** | **98.42%** | **100%** | **97.93%** |
| **0.5** | **98.06%** | **100%** | **98.62%** |

**Regularization Parameter = 0.001**

Loss/Error

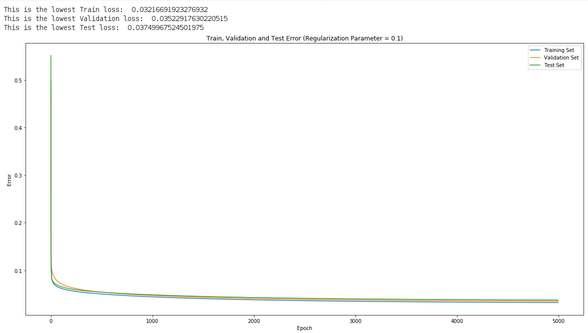


Accuracy

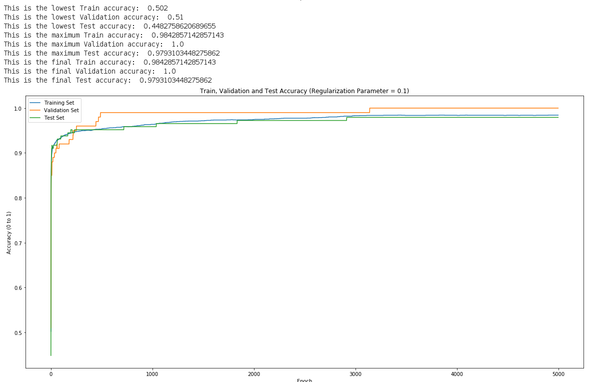


**Regularization Parameter = 0.1**

Loss/Error

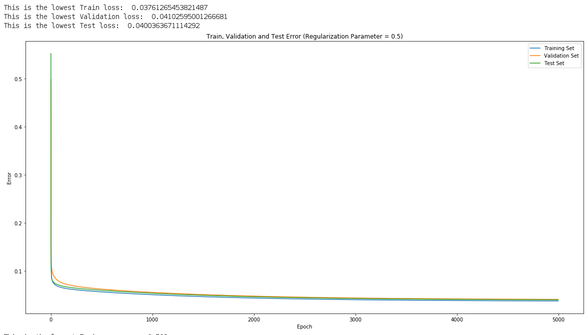


Accuracy

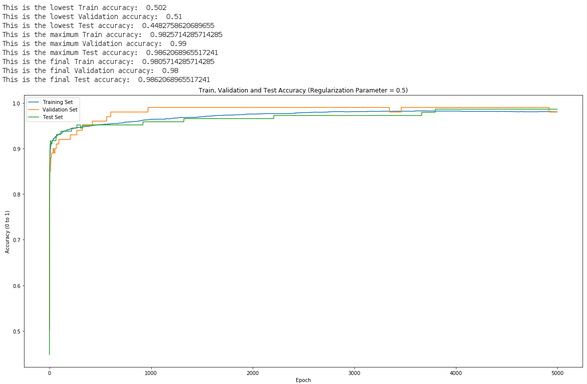


**Regularization Parameter = 0.5**

Loss/Error



Accuracy



## 5) Comparing Batch GD with normal equation [2 pts]:

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.

### Answer:

**Normal Equation**

We created a function to find the optimal W by using the normal equation. We also set a timer inside this function to calculate the time it takes to come up with the optimal W. We also calculated the accuracy of this optimal W using MSE. From our function we found out this function takes **0.175s** and has an accuracy of **99.76%**.

**Batch GD**

From Section 1.4, we can tell that the Final Training Accuracy is **98.6%**. Keep in mind that the inputs were bais = 1, learning rate = 0.005, epochs = 5000, regularization parameter = 0.001. We had to play around with the grad\_descent function to find the time it takes for the Batch GD to find the optimal W. The time it takes to complete the grad\_descent function with the inputs described earlier is **166.41s**.

"""

x is the data

y is the labels

"""

def leastSquares(x, y, b, reg):

# Start the timer

start = time.time()

# Reshape the matrix and find the transpose of the new matrix

x\_Orig = x.reshape((x.shape[0], x.shape[1] \* x.shape[2]))

b\_ones = np.ones((x.shape[0], 1))

# Add the one bias vector to all the weights

newX = np.concatenate((b\_ones, x\_Orig), axis = 1)

xTranspose = newX.transpose()

# Calculate the the error using thid formula W = [(X.T \* X)^-1 \*(X.T \* Y)]

e\_in = np.linalg.inv(np.matmul(xTranspose, newX))

wStar = np.matmul(e\_in, np.matmul(xTranspose, y))

b = wStar[0][0]

# Now we need to remove the bias column

wStar = wStar[1:].squeeze()

# End the timer

end = time.time()

totalTime = (end - start)

# Find the Accuracy

accuracyLeastSquares = accuracy(wStar, b, x, y, 'MSE')

print("This is the accuracy of the normal equation: ",accuracyLeastSquares)

print("This is the time it took to complete this computation: %.3lfs" % totalTime)

return wStar, b

# Section 2: Logistic Regression [10 points]

## 1) Loss Function and Gradient [4 pts]:

Implement two vectorized Numpy functions to compute the Binary Cross Entropy Error and its gradient respectively. Similar to Part 1.1, both functions should accept 5 arguments - the weight vector, the bias, the data matrix, the labels, and the regularization parameter. They should return the loss and gradients with respect to weights and bias respectively. Include the analytical expressions in your report as well as a snippet of your Python code.

### Answer:

"""

calculates the cross entropy and returns the total loss

W is a weight, 1D-array of size: 784 (28 x 28)

b is the bias

x is the batch of Mnist data images : (3500, 28, 28)

y is the batch of labels : (3500, 1)

reg is λ, the regularization parameter

"""

def crossEntropyLoss(W, b, x, y, reg):

# Your implementation here

#reshaping the batch to satisfy the size of weights for multiplication

#(x shape = (3500, 784), W shape = (784, 1))

x\_flat = x.reshape((x.shape[0], x.shape[1] \* x.shape[2]))

#for sigmoid func (prediction matrix shape = (3500, 1))

prediction\_matrix = np.dot(x\_flat, W) + b

#obtain model output (finding values between 1 and 0)

model\_output = sigmoid(prediction\_matrix)

#calculate error using the formula

left\_term = np.multiply(-1\*y, np.log(model\_output))

right\_term = np.multiply(1 - y, np.log(1 - model\_output))

ce = (left\_term - right\_term).mean()

#calculating weight decay loss

weight\_loss = (reg / 2) \* (np.linalg.norm(W) \*\* 2)

#return the sum of the two losses for overall loss

return weight\_loss + ce

def gradCE(W, b, x, y, reg):

# Your implementation here

# Reshaping part of the code

newW = W.reshape((x.shape[1] \* x.shape[2], 1)) # New shape of W is [784, 1]

newX = x.reshape((x.shape[0], x.shape[1] \* x.shape[2])) # New shape of x is [3500, 784]

#Version 2

length = x.shape[0]

#Find our prediction

x\_Prediction = np.dot(newX, newW) + b

#Obtain model output (finding values between 1 and 0)

model\_output = sigmoid(x\_Prediction)

e\_in = model\_output - y

#Find the gradient with respect to the weight

g\_Gradient = (np.dot(newX.transpose(), e\_in))/length

#Add the regularization factor

g\_Gradient = g\_Gradient + reg \* newW

#Find the gradient with respect to the bias

b\_Gradient = (np.sum(e\_in))/ length

return (g\_Gradient, b\_Gradient)

## 2) Learning [4 pts]:

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 bias using the Binary Cross Entropy loss and report on the performance of the Logistic Regression model by setting regularization parameter = 0.1, learning curve = 0.005, and 5000 epochs. Plot the loss and accuracy curves for training, validation, and test data set.

### Answer:

We already implemented this feature in Section 1 Part 2. We added a commented version of that code in the section down below.

"""

def grad\_descent(W, b, x, y, alpha, epochs, reg, error\_tol, lossType):

trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()

train\_loss = []

test\_loss = []

valid\_loss = []

train\_acc = []

valid\_acc = []

test\_acc = []

for i in range(epochs):

train\_acc.append(accuracy(W, b, x, y, lossType))

valid\_acc.append(accuracy(W, b, validData, validTarget, lossType))

test\_acc.append(accuracy(W, b, testData, testTarget, lossType))

#find loss and gradient weight and bias determined by input of this function

if lossType == 'MSE':

loss = MSE(W, b, x, y, reg)

train\_loss.append(loss)

valid\_loss.append(MSE(W, b, validData, validTarget, reg))

test\_loss.append(MSE(W, b, testData, testTarget, reg))

weights, bias = gradMSE(W, b, x, y, reg)

elif lossType == 'CE':

loss = crossEntropyLoss(W, b, x, y, reg)

train\_loss.append(loss)

valid\_loss.append(crossEntropyLoss(W, b, validData, validTarget, reg))

test\_loss.append(crossEntropyLoss(W, b, testData, testTarget, reg))

weights, bias = gradCE(W, b, x, y, reg)

#if error is smaller than the tolerance, we have the optimal values

#for weights and bias, thus we can stop iterating

if loss <= error\_tol:

break

#update the weight and bias from calculated gradient

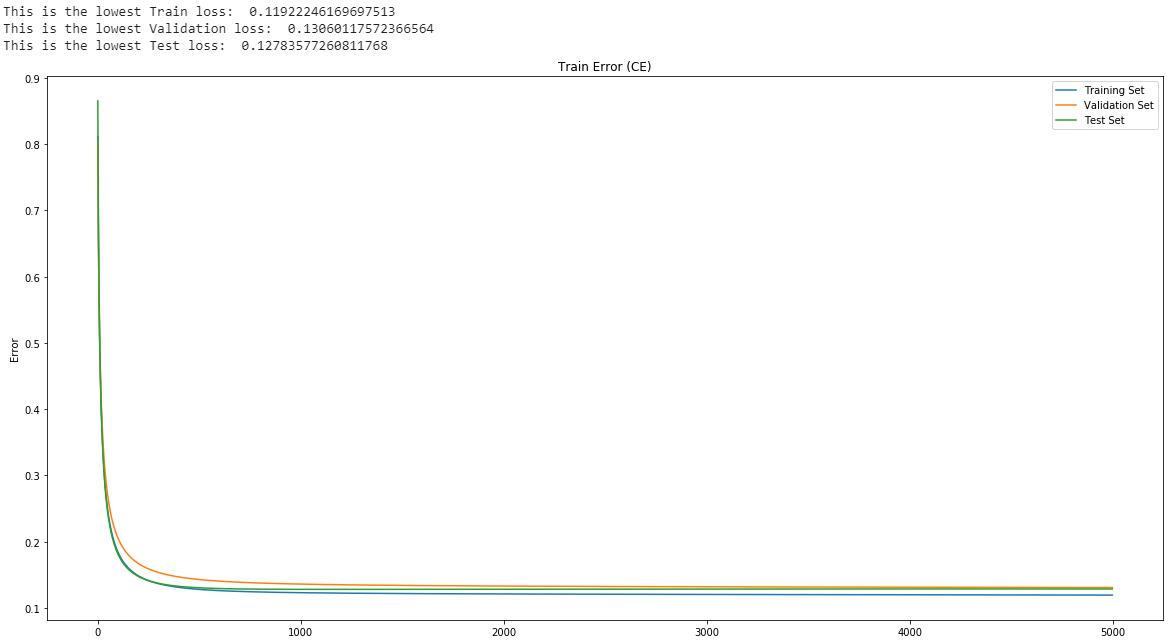
W -= alpha \* weights

b -= alpha \* bias

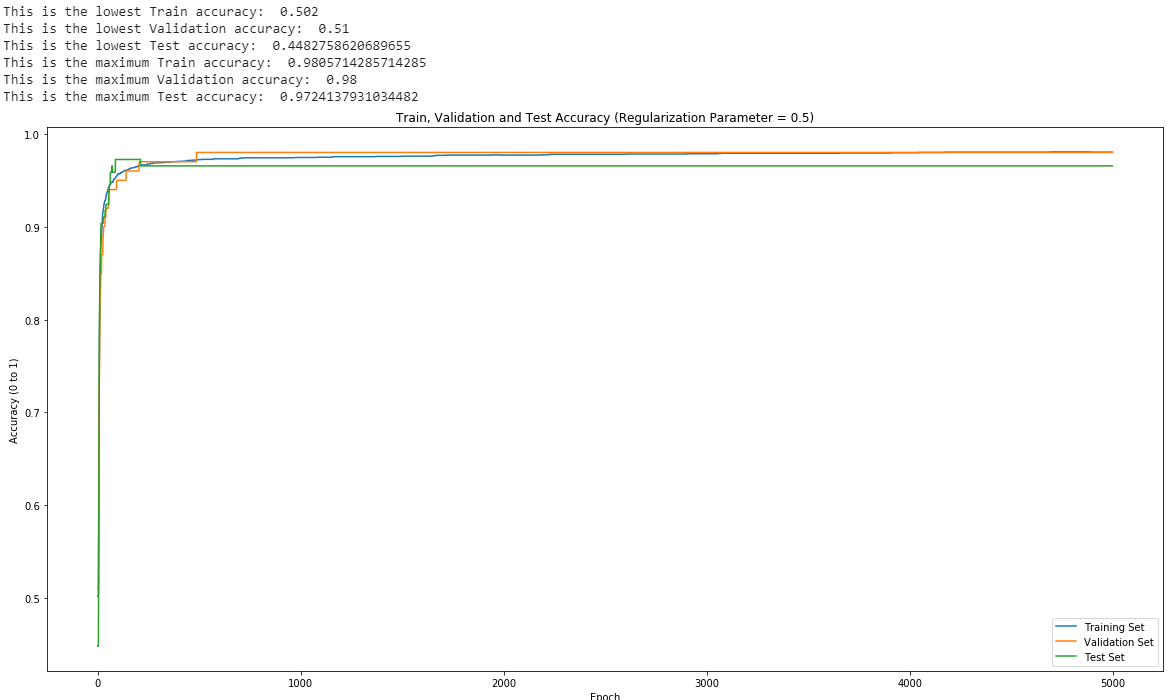
return W, b, train\_loss, valid\_loss, test\_loss, train\_acc, valid\_acc, test\_acc

"""

Loss/Error



Accuracy

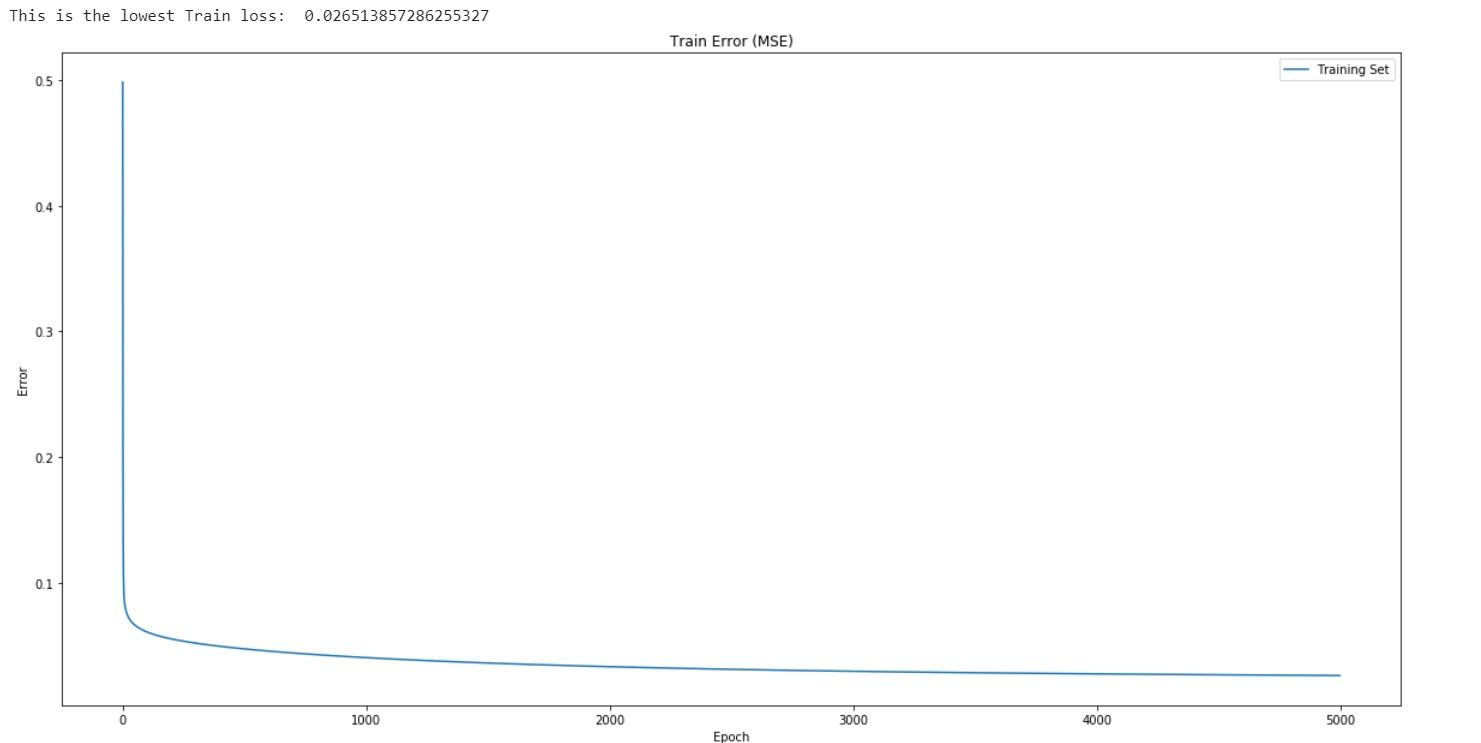
**3) Comparison to Linear Regression [2 pts]:**

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.

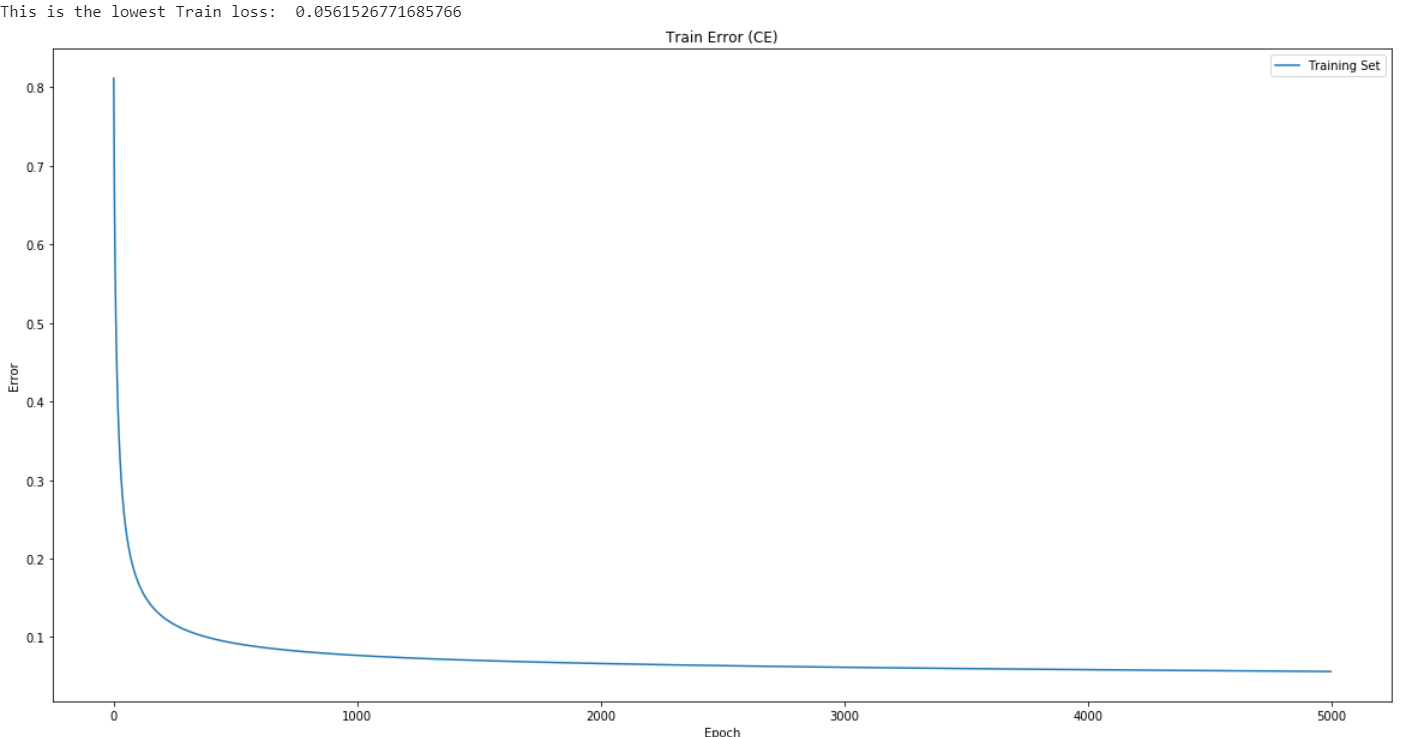
### Answer:

From the loss function of cross-entropy we can easily tell that the logistic regression model using cross-entropy converges faster than linear regression model using MSE. This tells us that cross-entropy is a better at measuring the loss compared to MSE for classification problems. In our case just by looking at the graph we can tell this does not happen.

**MSE Graph**

****

**Cross-entropy Graph**

****

**Section 3: Batch Gradient Descent vs. SGD and Adam [25 points]**

**1) Building the Computational Graph [5 pts]:**

Build a function, buildGraph() that accepts one argument, the loss type (either MSE or

CE) and initializes the Tensorflow computational graph. To do so, you must initialize the

following:

(a) The weight and bias tensors (for the weight tensors, use the tf.truncated normal com-

mand and set the standard deviation to 0.5.

(b) Tensors to hold the "variables"; use the tf.placeholder command (e.g. tensors for the

data, labels and regularization parameter.)

(c) The loss tensor. Depending on the parameter passed to the buildGraph() function, you

will need to either create the MSE or CE loss function with the regularization parameter. You may wish to investigate the Tensorfow API Documentation regarding losses and regularization on their website.

(d) The optimizer. Be sure to set it to minimize the total loss. Set to 0.001.

The function should return the Tensorflow objects for weight, bias, predicted labels, real

labels, the loss, and the optimizer. The function header is below.

**Answer:**

def buildGraph(loss, b1, b2, eps):

#Initialize weight and bias tensors

tf.set\_random\_seed(421)

input\_size = 28\*28

W = tf.Variable(tf.truncated\_normal(stddev=0.5, shape=(input\_size, 1), dtype=tf.float32))

b = tf.Variable(name='bias', initial\_value = tf.ones(1), dtype=tf.float32)

x = tf.compat.v1.placeholder(dtype=tf.float32, shape=(None, input\_size))

y = tf.compat.v1.placeholder(dtype=tf.int8, shape=(None))

reg = tf.placeholder(dtype=tf.float32, name='lambda')

predicted\_y = tf.add(tf.matmul(x, W), b)

optimizer = tf.train.AdamOptimizer(learning\_rate = 0.001, beta1=b1, beta2=b2, epsilon=eps)

if loss == "MSE":

# Your implementation

err = tf.losses.mean\_squared\_error(y, predicted\_y)

elif loss == "CE":

#Your implementation here

predicted\_y = tf.sigmoid(predicted\_y)

err = tf.losses.sigmoid\_cross\_entropy(y, predicted\_y)

optimal = optimizer.minimize(err)

return W, b, x, predicted\_y, y, err, optimal

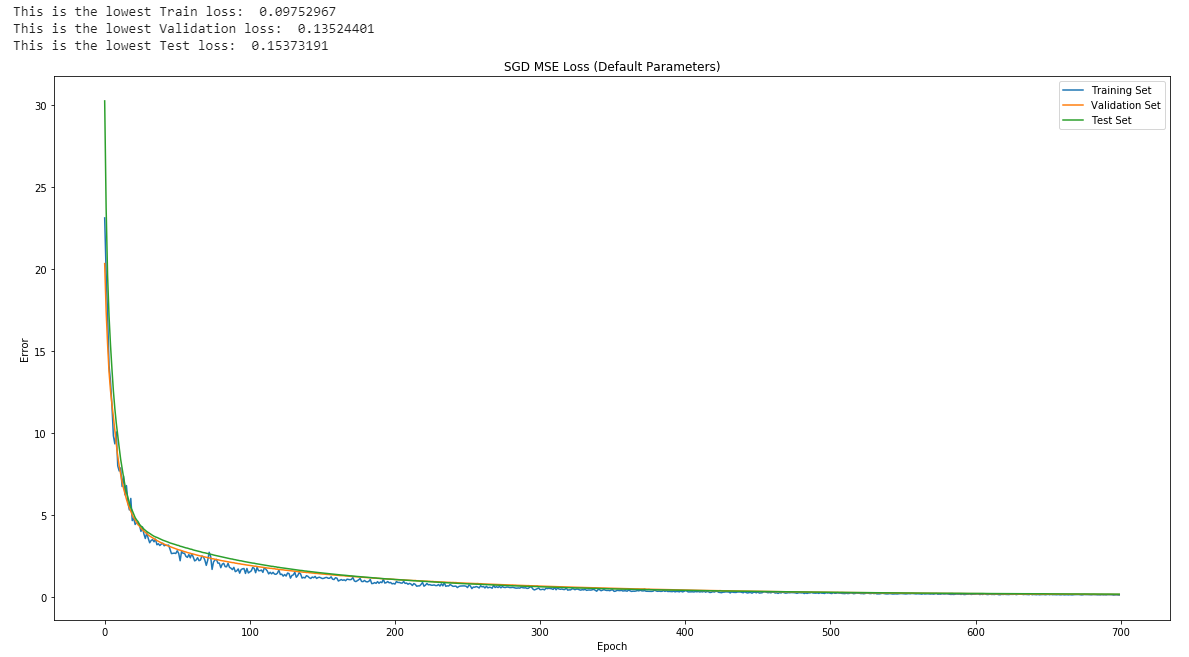
**2. Implementing Stochastic Gradient Descent [5 pts]**

Implement the SGD algorithm for a minibatch size of 500 optimizing over 700 epochs, minimizing the MSE (you will repeat this for the CE later). 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 = 0 and continue to use the same learning rate value (i.e. 0.001). After each epoch, store the training, validation and test losses and accuracy. Use these to plot the loss and accuracy curves.

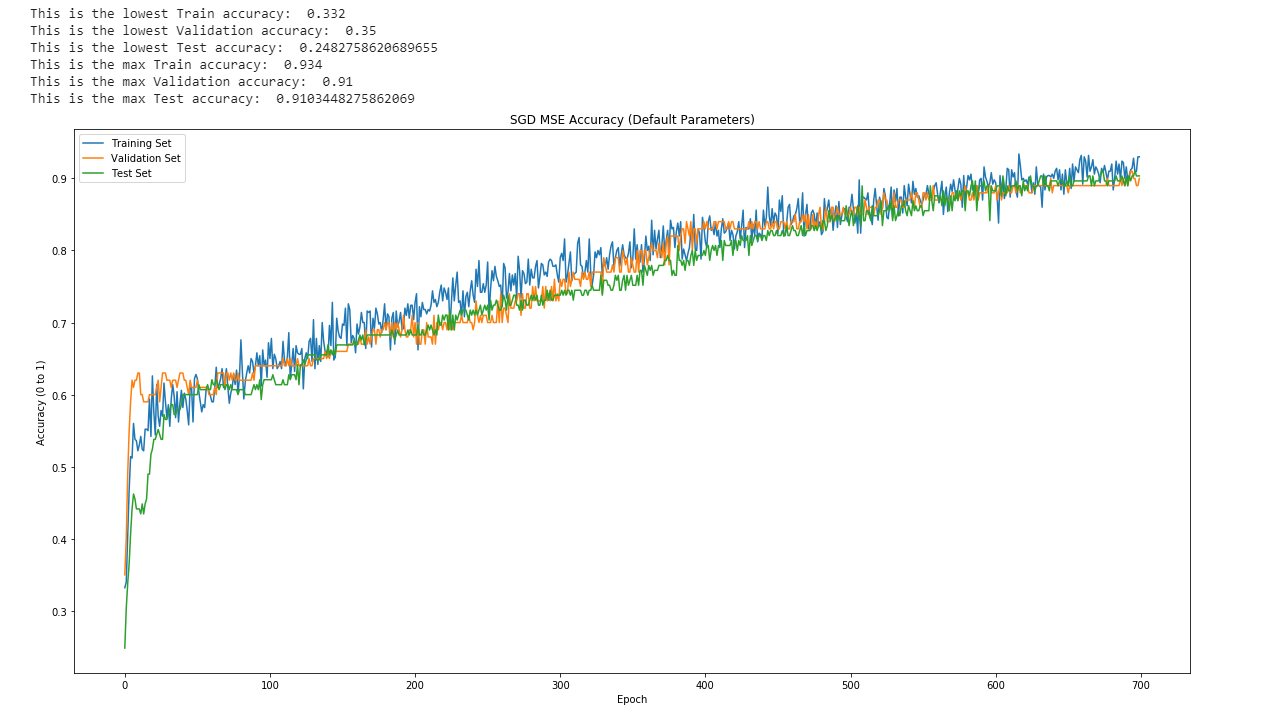
**Answer:**

**MSE Graphs**

Loss/Error



Accuracy



**3. Batch Size Investigation [2 pts]**

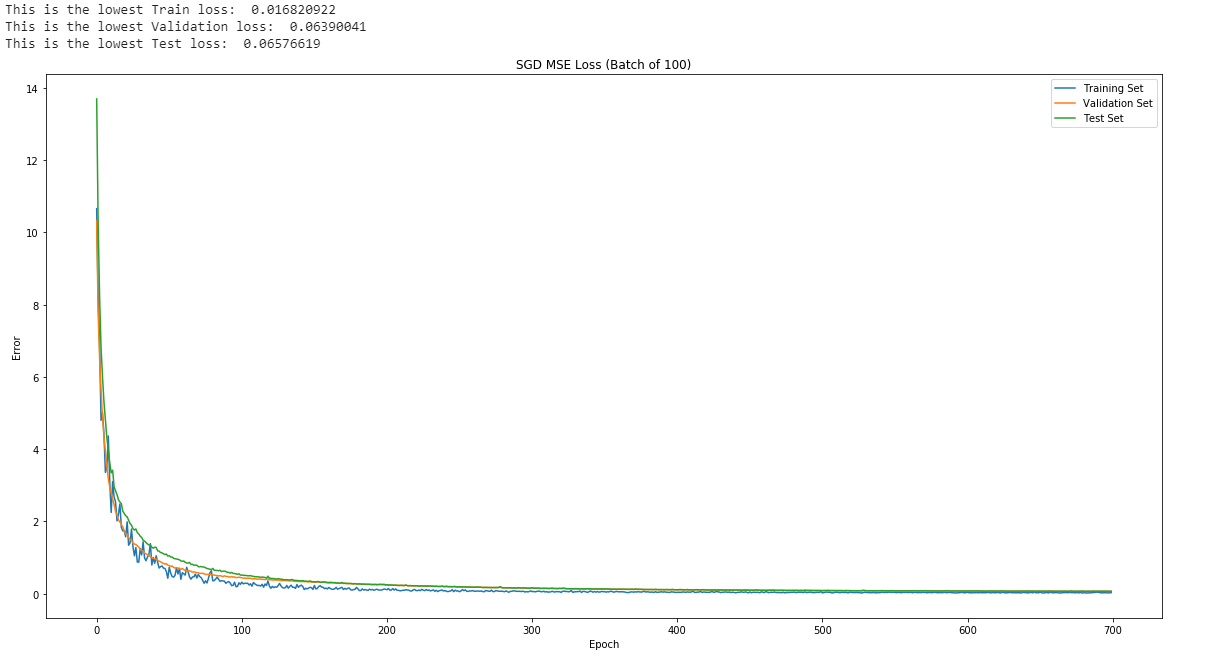
Study the effects of batch size on behaviour of the SGD algorithm optimized using Adam by optimizing the model using batch sizes of B ={100; 700; 1750}. For each batch size, plot training/validation/test MSE loss in one plot and training/validation/test accuracy in another plot. (you need to have a total of 6 plots for this section). What is the impact of batch size on the final classification accuracy for each of the 3 cases? What is the rationale for this?

**Answer:** (MSE graphs are located at 3.5)As we reduce the batch size, the error and loss converges much faster to an optimal solution. As shown for both MSE and CE, as the batch size is reduced, the accuracy rises close to 100% and the loss reduces at a faster rate than the larger batch size. The AI makes the weights more accurate for the smaller batch size which results in more accuracy for that specific batch since the gradient is updated much more frequently. You can see this occur in the CE batch 100 test. The weights are actually being overfitted for the data as the optimal is reached in a small number of epochs. The loss for the validation and test set increases. This is because the AI has a chance to look and set appropriate weights for a small batch more accurately with less computations, resulting in overfitting the data. Meanwhile with larger batches there are greater amounts of data points. This results in the weights being less accurate for the batch since there are so many data points the weights must satisfy and thus requires more epochs to determine the optimal weights. This is shown in the MSE graphs, the accuracy is much lower (61.3%) for the 1750 batch size meanwhile, with a batch size of 100, the accuracy increases by 34% (95.2%).

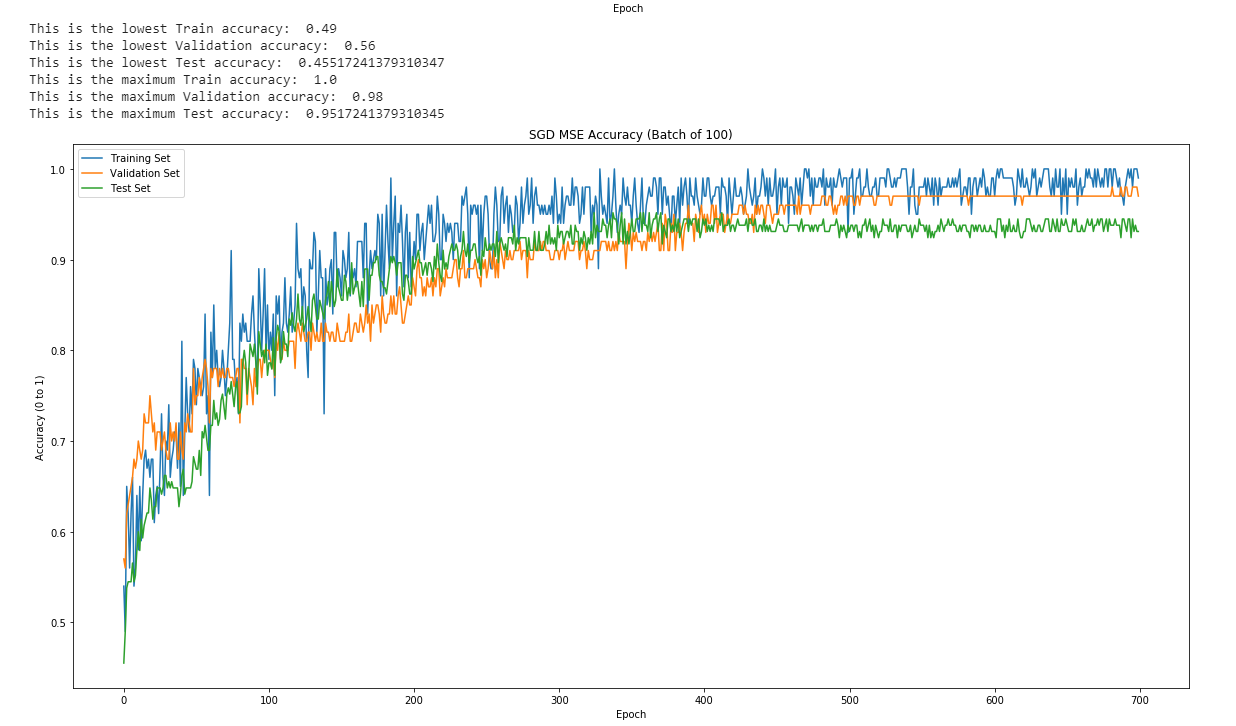
**MSE Graphs**

**Batch Size = 100**

Loss/Error

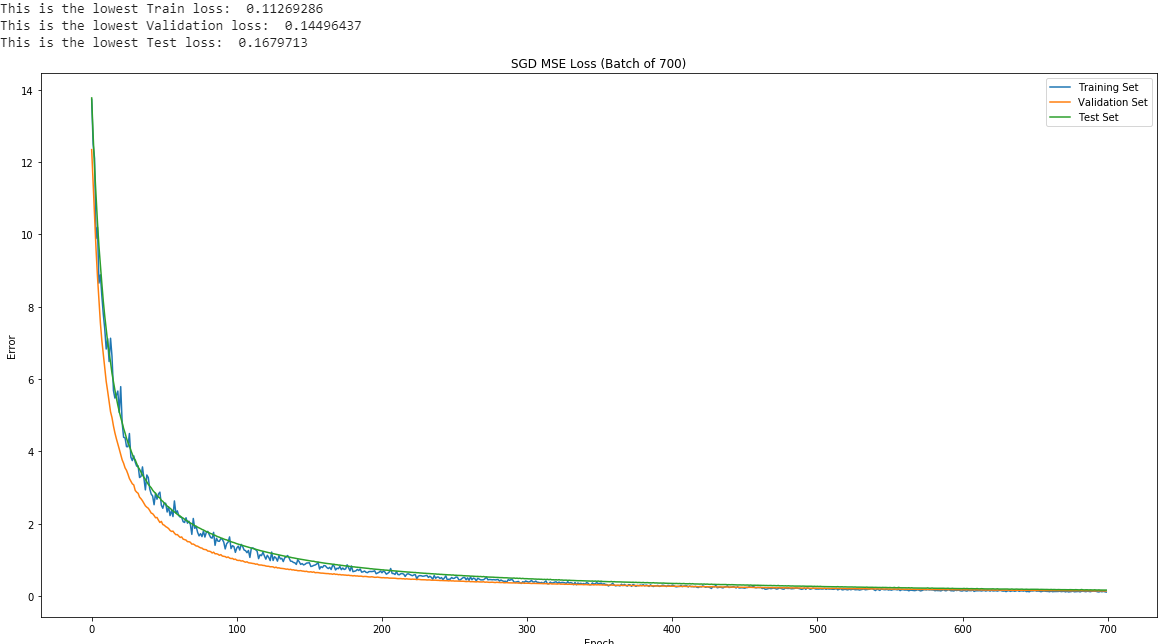


Accuracy

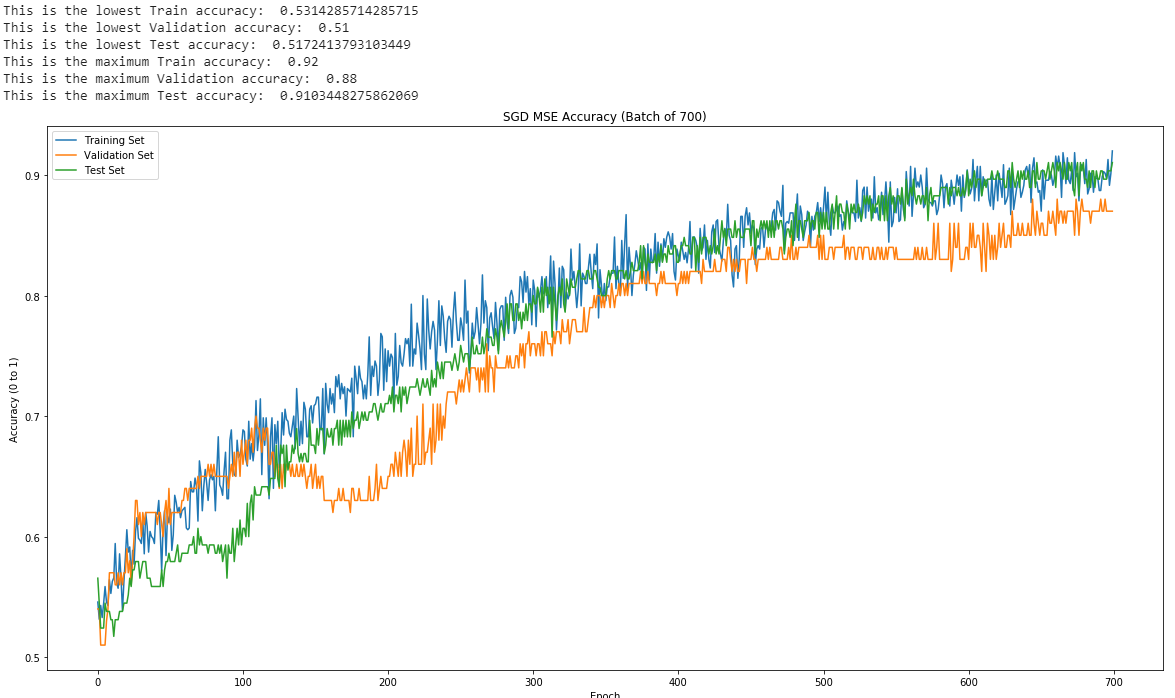


**Batch Size = 700**

Loss/Error

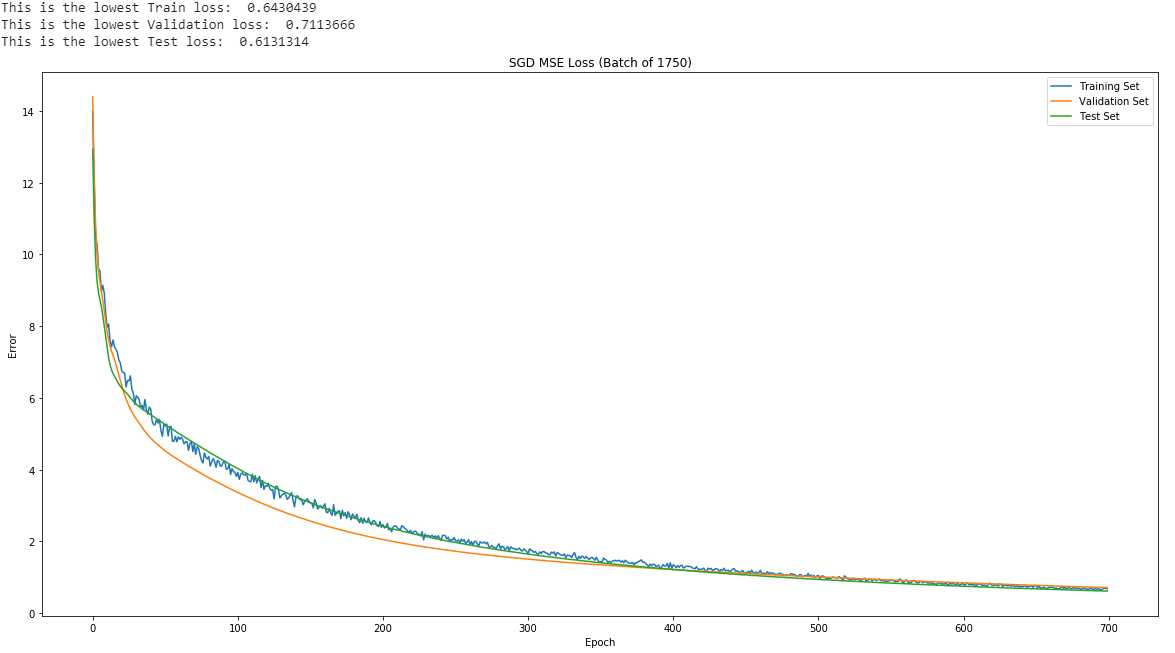


Accuracy

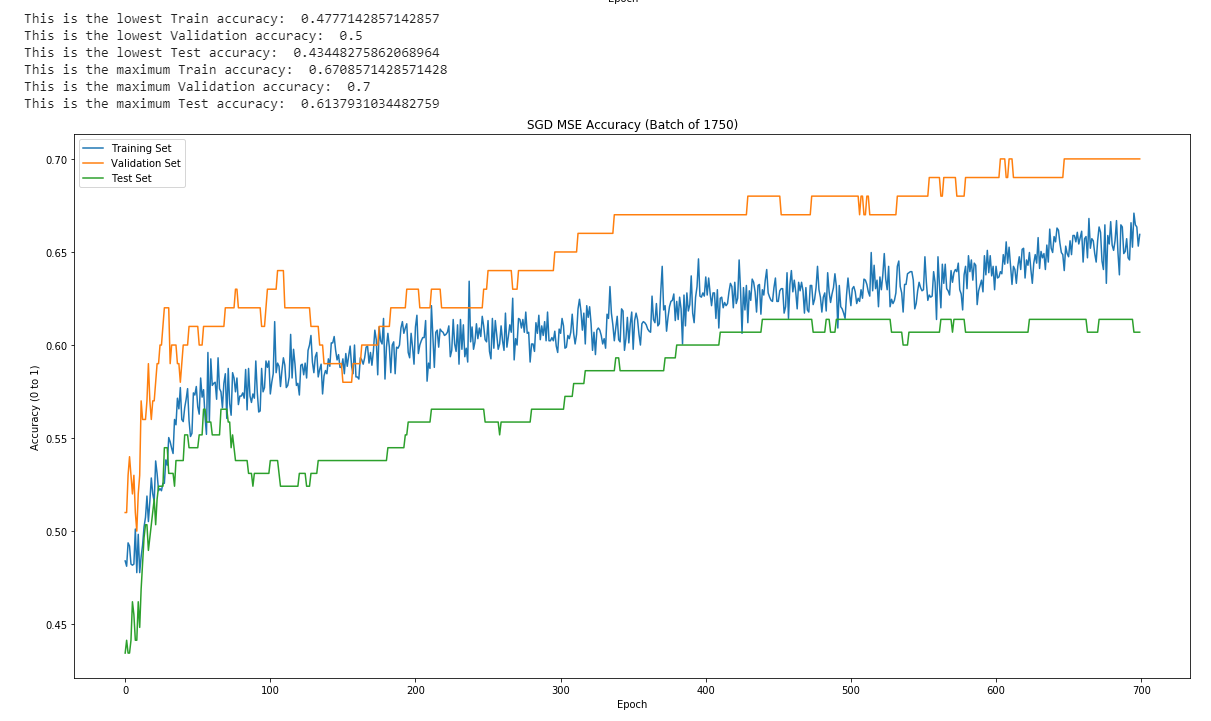


**Batch Size = 1750**

Loss/Error

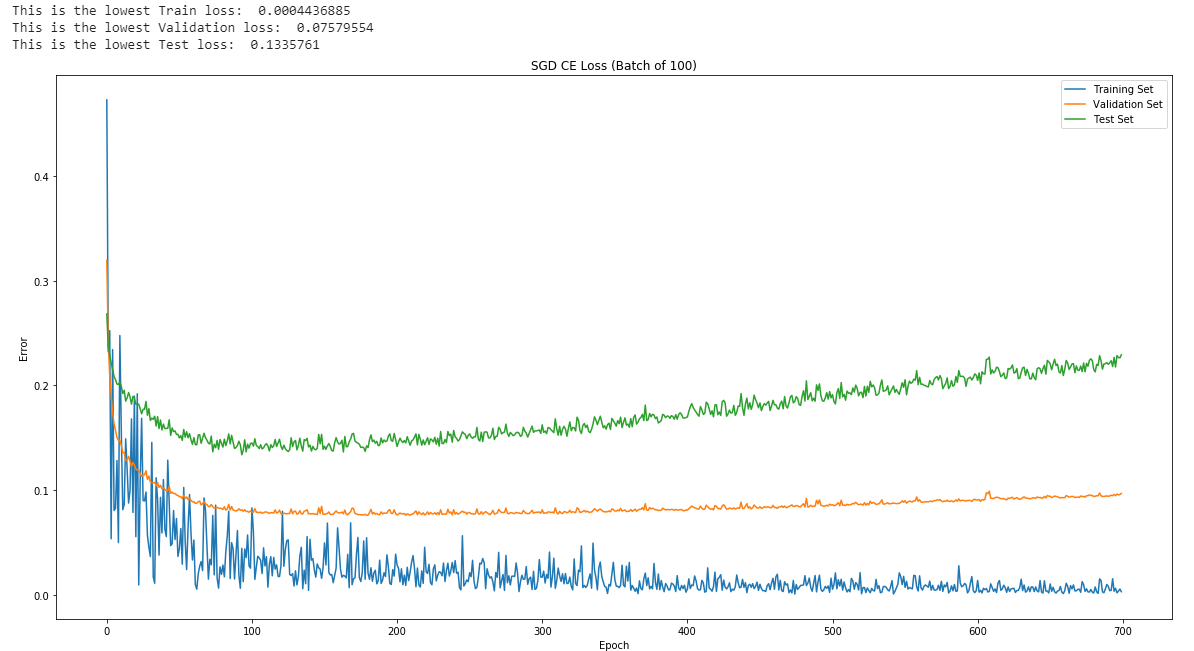


Accuracy

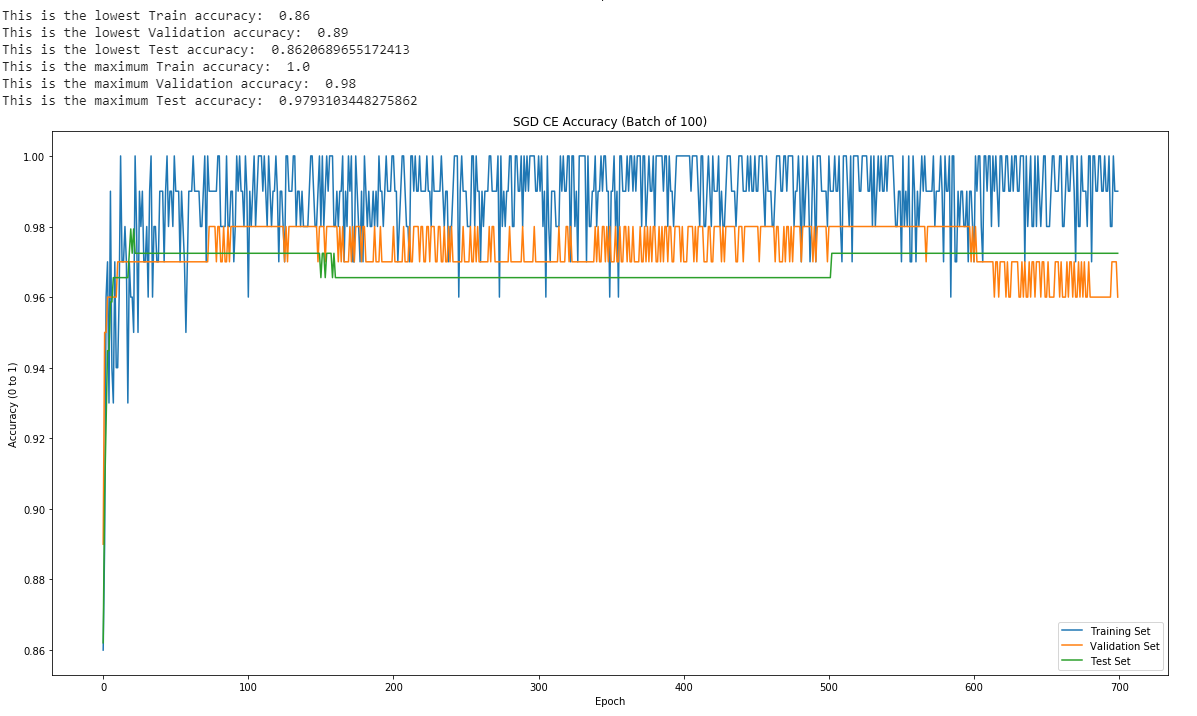


**CE Graphs**

**Batch Size = 100**

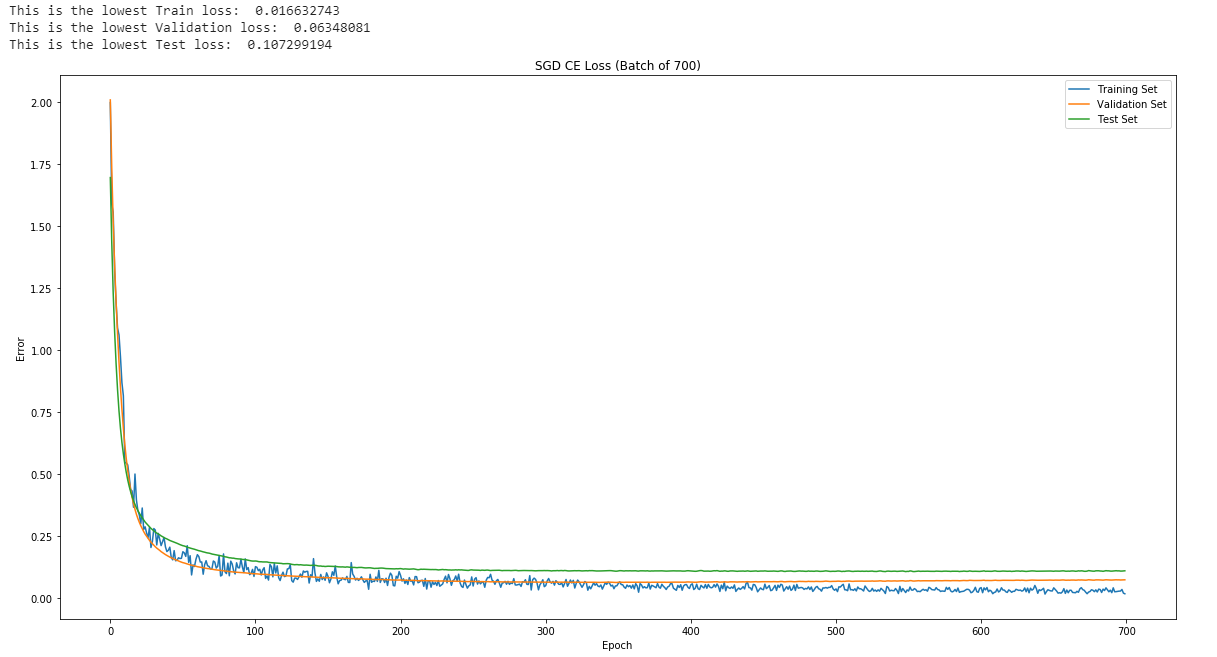


Accuracy

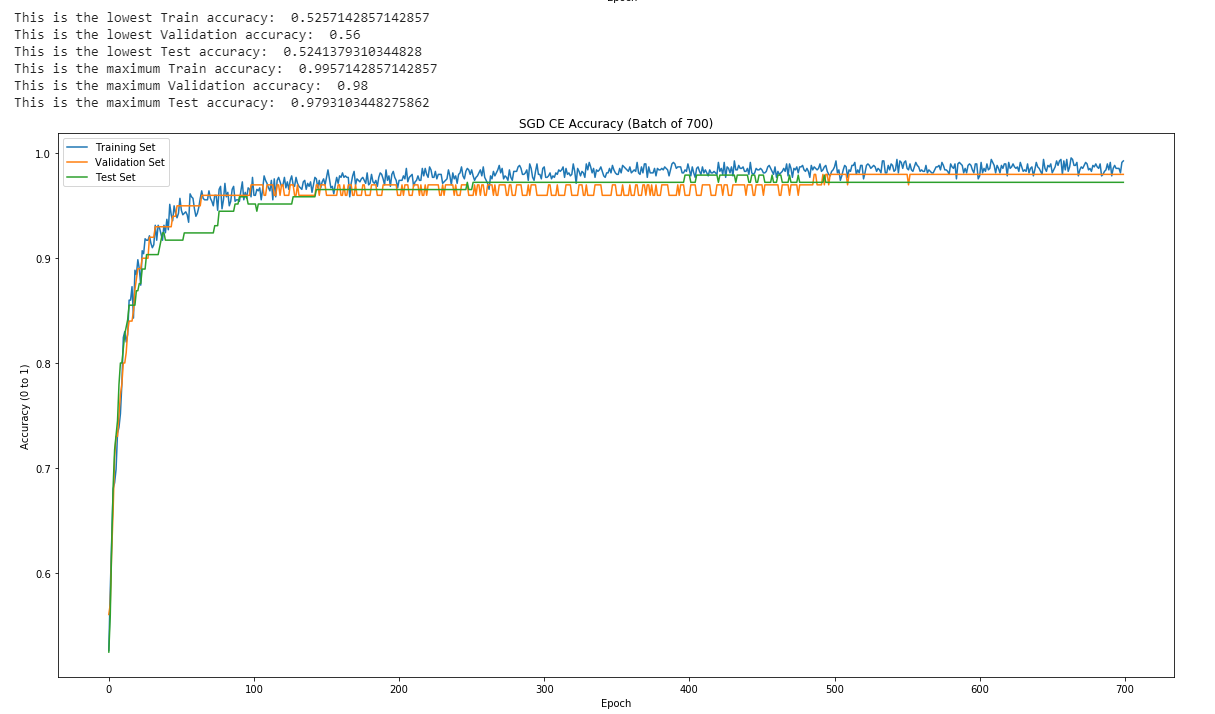


**Batch Size = 750**

Loss/Error

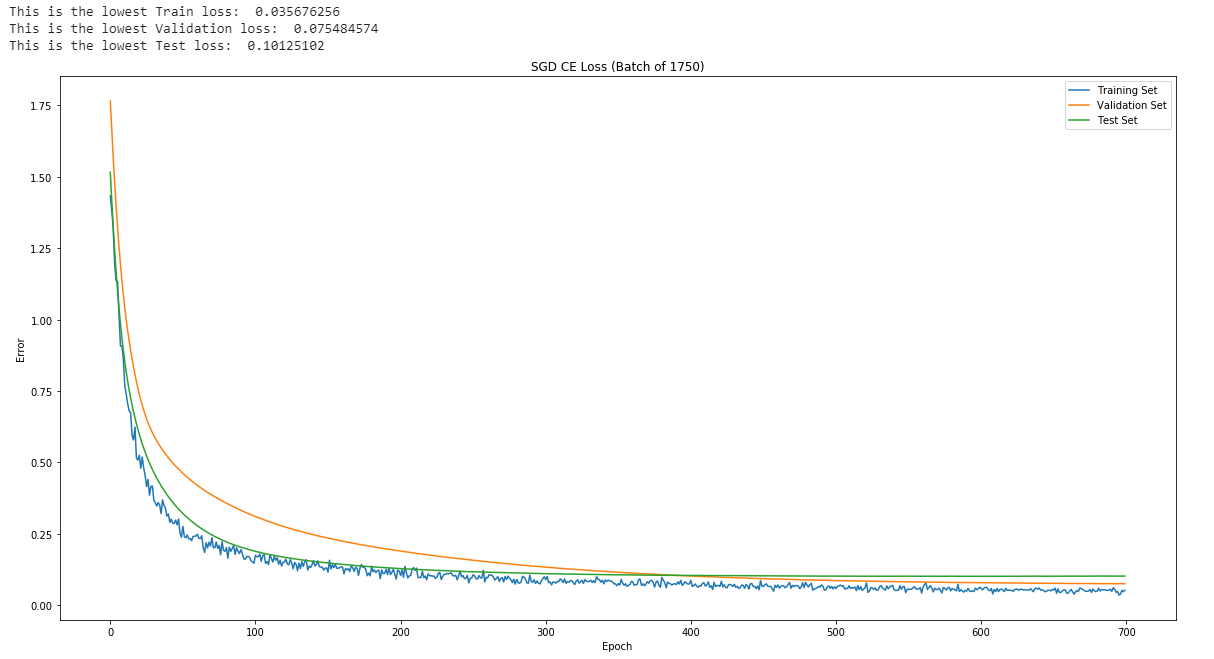


Accuracy

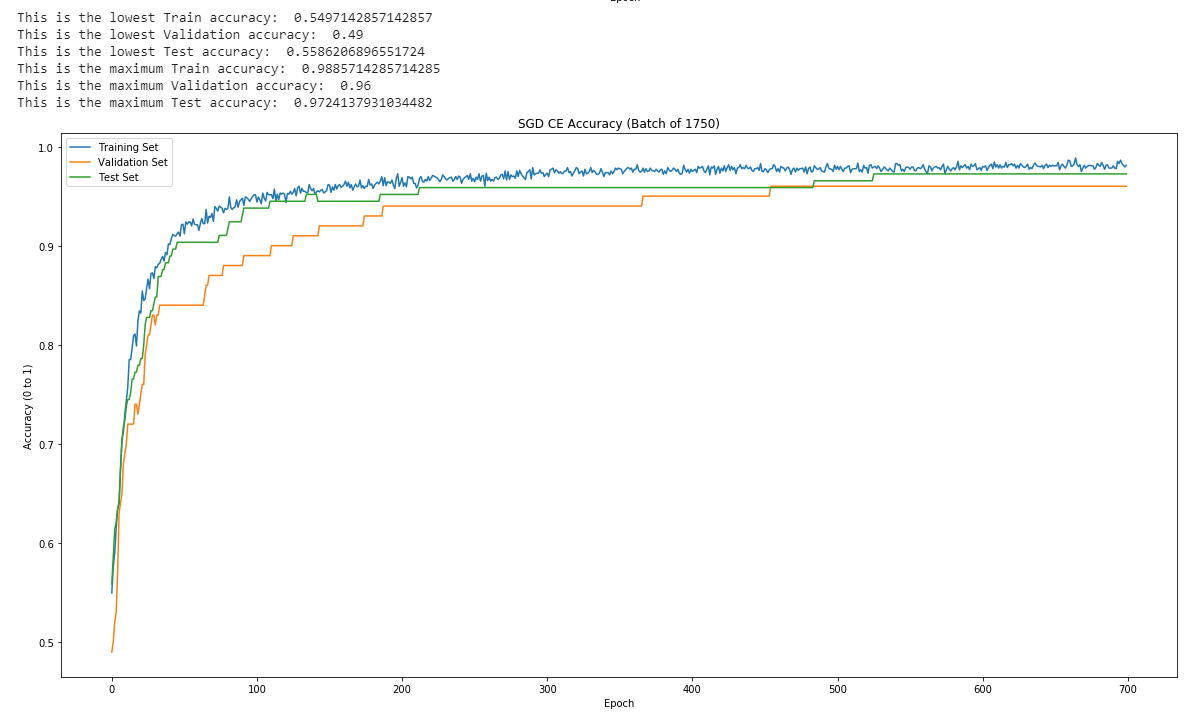


**Batch Size = 1750**

Loss/Error



Accuracy



**4. Hyperparameter Investigation [4 pts.]**

Experiment with the following Adam hyperparameters and for each, report on the nal training, validation and test accuracies:

(a) 𝜷1 = {0:95, 0:99}

(b) 𝜷2 = {0:99, 0:9999}

(c) 𝞮 = {1e - 09, 1e - 4}

For this part, use a minibatch size B = 500 and a learning rate of 𝜶 = 0:001 and optimize

over 700 epochs. For each of the three hyperparameters listed above, keep the other two as

the default Tensorflow initialization. Note that in order to set 1, 2, and , you may wish

to add these parameters to build\_graph() inputs. For each, what is the hyperparameter impact on the final training, validation and test accuracy? Why is this happening?

**Answer:** When increasing the value in beta1 and beta1, the MSE accuracy begins to decrease. When epsilon approaches a larger number furthering from 0, the accuracy begins to decrease significantly.

Beta1 and 2 determine how much the learning rate decays. Adam uses the estimation of the first and second moment respectively to determine how much the weight should change. Beta 1 takes into consideration the historical values of momentum in Adam, thus, a greater value of beta would cause the optimal weight to converge slower as the prior value of momentum has a greater impact on the new weight it calculates. Therefore, as the error approaches the global minimum, the added value of momentum would skip the minimum more times before reaching it. This is a similar idea with beta 2. As beta 2 increases, the past momentum is factored in with greater rations, decreasing the accuracy overall.

The epsilon value makes sure that when updating the weight values, no value is divided by 0. Thus when increasing the epsilon value, the weight update is changed by a smaller value (as dividing by a larger value results in a smaller value). Therefore, as the epsilon approaches a larger value, the change in weight is much smaller, reducing the accuracy when the number of iterations remain the same.

For CE, the value stays relatively the same for any change in the parameters. This may be because CE reaches the optimal point much faster compared to MSE. Therefore, for larger values of epoch, the beta and epsilon values did not change the accuracy. However, for the early stages of training, the values may vary drastically since the beta and epsilon values heavily influence the learning rate and its momentum.

|  |  |  |  |
| --- | --- | --- | --- |
| **New Variable**  **(MSE)** | **Final Training Accuracy** | **Final Validation Accuracy** | **Final Testing Accuracy** |
| **Default** | **87.20%** | **82.0%** | **83.45%** |
| **𝜷1 = 0.95** | **64.91%** | **61.0%** | **70.34%** |
| **𝜷1 = 0.99** | **63.89%** | **63.0%** | **63.45%** |
| **𝜷2 = 0.99** | **85.77%** | **83.0%** | **84.14%** |
| **𝜷2 = 0.9999** | **76.23%** | **76.0%** | **70.34%** |
| **𝞮 = 1e-09** | **92.20%** | **88.0%** | **88.28%** |
| **𝞮 = 1e-04** | **69.54%** | **64.0%** | **66.90%** |

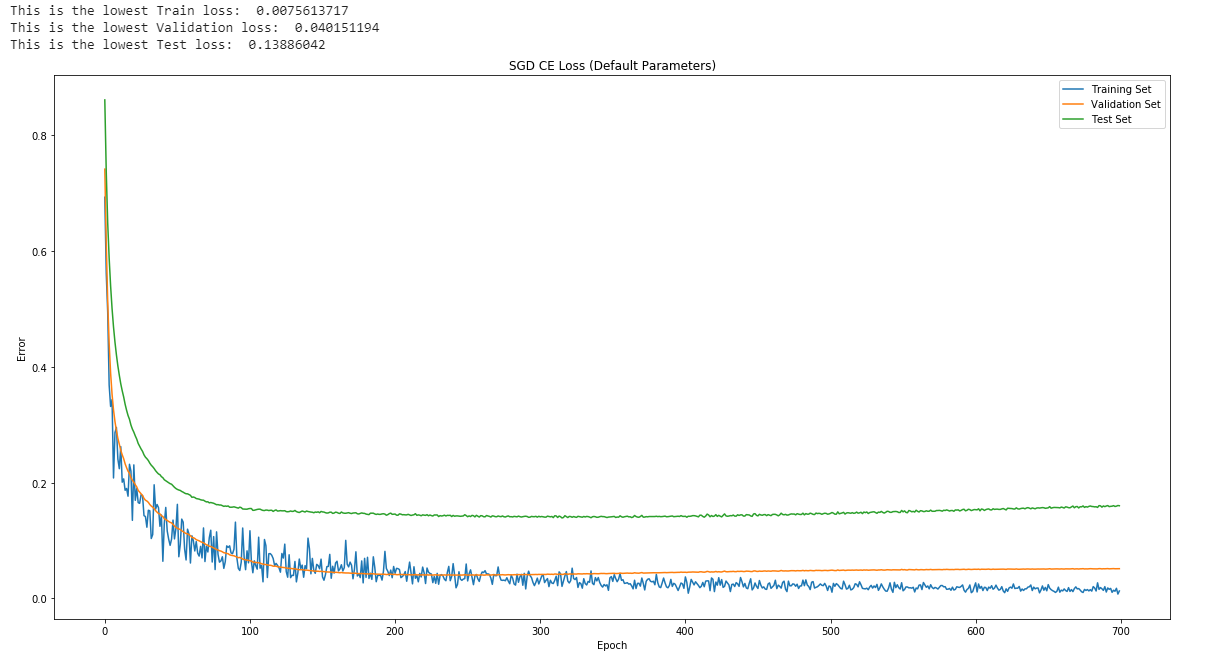
**5) Cross Entropy Loss Investigation [6 pts.]**

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)?

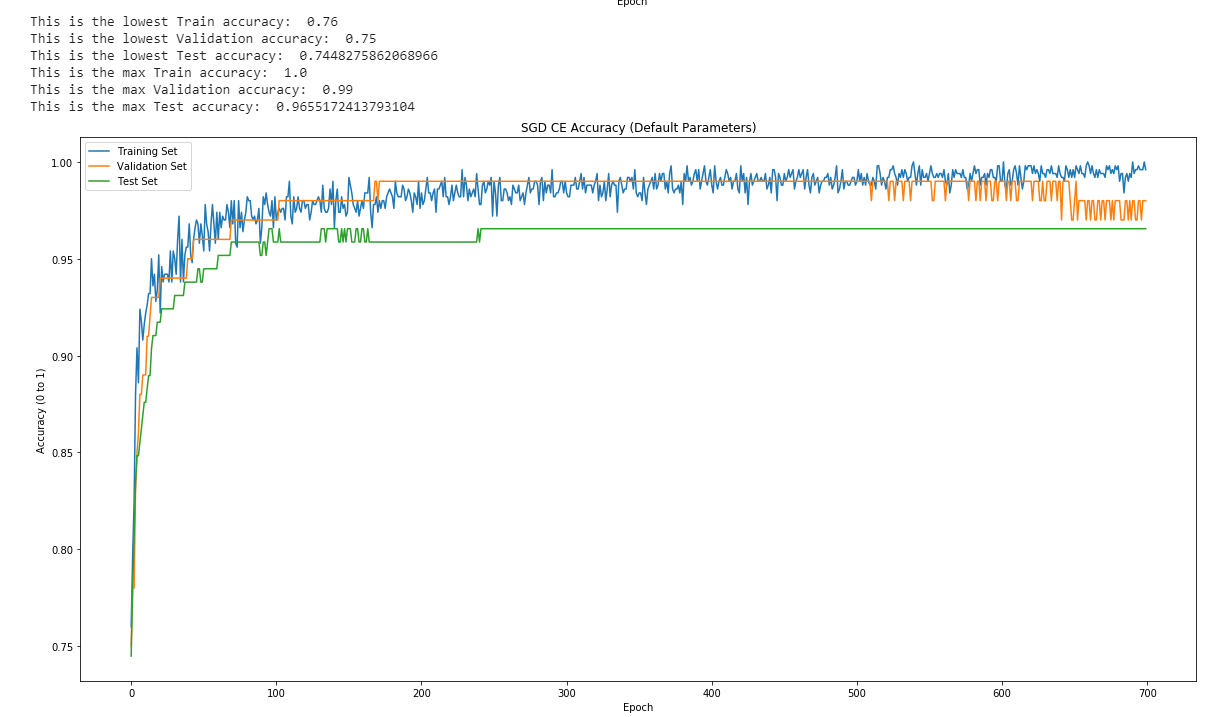
**Answer:** Binary cross entropy is a much better loss function for final classification accuracy. For the different batch sizes, the BCE reaches an accuracy over 90% quickly while MSE has trouble with gaining higher values of accuracy. Even while changing the hyperparameters for part 5.4, the accuracy of BCE seemed to significantly overtake MSE: MSE having their highest values near 85% while BCE all consistently having a percentage over 95%. Moreover, the accuracy values obtained through CE seem much more consistent overall, showing that CE converges at a much faster pace thus not requiring any optimization with the parameters.

**3.12 CE Graphs**

Loss/Error



Accuracy



**3.14**

|  |  |  |  |
| --- | --- | --- | --- |
| **New Variable**  **(CE)** | **Final Training Accuracy** | **Final Validation Accuracy** | **Final Testing Accuracy** |
| **Default** | **99.40%** | **97.0%** | **96.55%** |
| **𝜷1 = 0.95** | **99.60%** | **97.0%** | **97.24%** |
| **𝜷1 = 0.99** | **99.60%** | **97.0%** | **97.24%** |
| **𝜷2 = 0.99** | **99.40%** | **97.0%** | **97.93%** |
| **𝜷2 = 0.9999** | **99.40%** | **96.0%** | **97.93%** |
| **𝞮 = 1e-09** | **99.0%** | **98.0%** | **98.62%** |
| **𝞮 = 1e-04** | **99.50%** | **97.0%** | **97.24%** |

**6) Comparison against Batch GD [3 pts.]**

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?

**Answer:**

The reason the accuracy is higher in gradient descent for MSE is because the number of epochs in gradient descent is much larger than the epochs used in SGD. As explained in 3.5, MSE converges slower, thus requiring many more epochs to acquire a greater accuracy. The gradient descent has more iterations of training which can cause the accuracy to go to 100 percent explaining the result we obtained. Although SGD is much more optimised due to the adaptive changes in learning rate and its ability to escape local minimums due to the momentum, the data does not reflect this effectively possible due to the number of epochs. Therefore, increasing the epoch values for SGD could potentially result in a similar outcome

The values for CE are much more similar. This is true due to the nature of CE. The rate at which CE converges is significantly fast that using either GD or SDG will result in a similar outcome. The loss values are slightly different because of the optimisation Adam uses to escape local minimums.

*Tables Below Based on these hyperparameters:*

*GD MSE & CE: regularization parameter = 0.1, learning curve = 0.005, and 5000 epochs*

*SGD MSE & CE: mini batch = 500, epochs = 700, learning rate = 0.001*

|  |  |  |  |
| --- | --- | --- | --- |
| **MSE** | **Training Set** | **Validation Set** | **Testing Set** |
| **GD Loss** | 0.0322 | 0.0352 | 0.0375 |
| **SGD Loss** | 0.0975 | 0.135 | 0.154 |
| **GD Accuracy (%)** | 98.4 | 100 | 97.9 |
| **SGD Accuracy (%)** | 93.4 | 91 | 91 |

|  |  |  |  |
| --- | --- | --- | --- |
| **CE** | **Training Set** | **Validation Set** | **Testing Set** |
| **GD Loss** | 0.119 | 0.131 | 0.128 |
| **SGD Loss** | 0.00756 | 0.0402 | 0.139 |
| **GD Accuracy (%)** | 98.1 | 98.0 | 97.2 |
| **SGD Accuracy (%)** | 100 | 99.0 | 96.6 |