3 Batch Gradient Descent vs SGD and ADAM

3.1 Building the computational graph

```
def buildGraph(losstype):
  tf.set random seed(421)

  # Initialize weight and bias tensors
  W = tf.Variable( tf.truncated normal(mean= 0.0, shape = (784, 1), stddev = 0.5, dtype =
tf.float32, name = "Weights") )
  B = tf.Variable( 0.0, name='Bias', dtype=tf.float32 )

  # Intialize data (X) and label (Y) placeholders
  X = tf.compat.v1.placeholder( tf.float32, shape=(None, 784), name="X" )
  Y = tf.compat.v1.placeholder( tf.float32, shape=(None, 1), name="Y" )
  Reg = tf.compat.v1.placeholder( tf.float32, shape=(), name="Reg" )

  # Model prediction
  Y hat = tf.linalg.matmul( X, W ) + B

  if losstype == "MSE":
    loss = tf.math.reduce mean( tf.math.squared difference(Y hat, Y) ) + (Reg/2)*tf.norm(W)

  elif losstype == "CE":
    loss = tf.math.reduce mean( tf.nn.sigmoid cross entropy with logits(labels=Y, logits=Y hat) ) +
(Reg/2)*tf.norm(W)

  # Optimizer
  Optimizer = tf.compat.v1.train.AdamOptimizer(0.001).minimize(loss)

  return W, B, X, Y hat, Y, Reg, loss, Optimizer
```

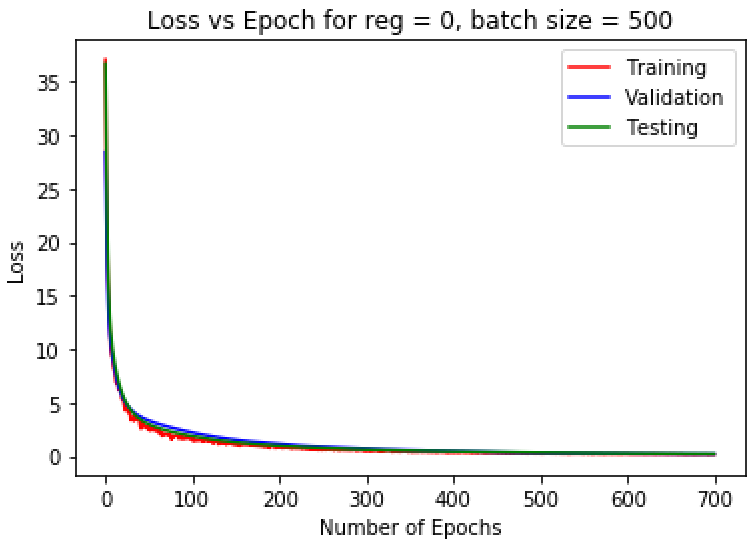
3.2 Implementing Stochastic Gradient Descent

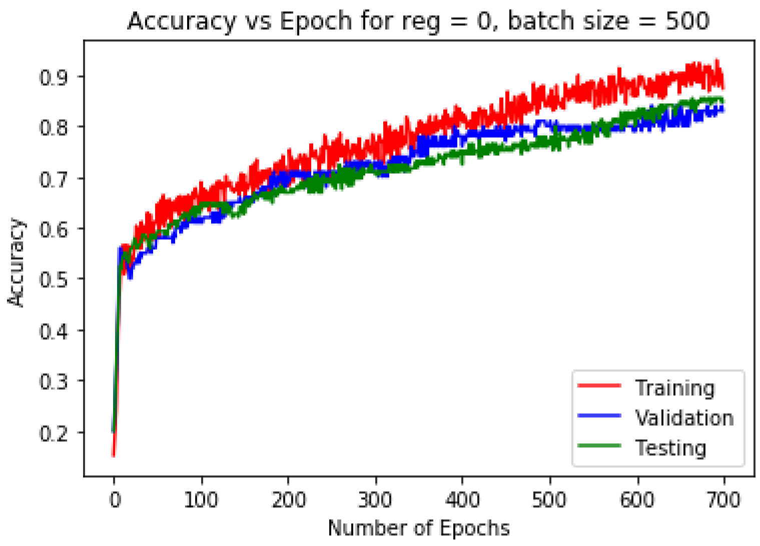
```
def TrainModel( trainData, validData, testData, trainTarget, validTarget, testTarget, reg,
losstype, epochs, batchSize ):

  # Store
  training loss = []
  valid loss = []
  test loss = []
  training acc = []
  valid acc = []
  test acc = []

  # Initialize tensors
  W, B, X, Y hat, Y, Reg, Loss, Optimizer = buildGraph(losstype)

  # launch session
  sess = tf.InteractiveSession()
  sess.run( tf.global variables initializer() )

  # Training
  for epoch in range(epochs):

    # Shuffle data and labels, make run graph
    shuffledData, shuffledTargets = Shuffle( trainData, trainTarget, epoch )

    # Train batch
    for batch in range( int(trainData.shape[0]/batchSize) ):
      batchData, batchTarget = makeBatch(shuffledData, shuffledTargets, batch, batchSize)
      w, b, y hat, loss, optimizer = sess.run( [W, B, Y hat, Loss, Optimizer],
feed dict={X:batchData, Y:batchTarget, Reg:reg} )

    w , b , y hat test, testloss = sess.run( [W, B, Y hat, Loss], feed dict={X:testData,
Y:testTarget, Reg:reg} )
    w , b , y hat valid, validloss = sess.run( [W, B, Y hat, Loss], feed dict={X:validData,
Y:validTarget, Reg:reg} )

    print(epoch)
    training loss.append( loss )
    test loss.append( testloss )
    valid loss.append( validloss )

    # Get accuracy
    acc = tf.compat.v1.placeholder( tf.float32, shape=(), name="acc" )
    validacc = tf.compat.v1.placeholder( tf.float32, shape=(2,1), name="validacc" )
    testacc = tf.compat.v1.placeholder( tf.float32, shape=(), name="testacc" )

    acc = sess.run( tf.math.reduce mean(tf.cast(tf.math.equal(tf.math.greater( y hat,
tf.constant(0.5, tf.float32, shape=(y hat.shape))), batchTarget), tf.float32)) )
    validacc = sess.run( tf.math.reduce mean(tf.cast(tf.math.equal(tf.math.greater( y hat valid,
tf.constant(0.5, tf.float32, shape=(y hat valid.shape))), validTarget), tf.float32)) )
    testacc = sess.run( tf.math.reduce mean(tf.cast(tf.math.equal(tf.math.greater( y hat test,
tf.constant(0.5, tf.float32, shape=(y hat test.shape))), testTarget), tf.float32)) )
    # print( sess.run([acc, validacc, testacc]) )

    training acc.append( acc )
    test acc.append( testacc )
    valid acc.append( validacc )

  Plot( reg, training loss, valid loss, test loss, epochs, batchSize, training acc, valid acc,
test acc )
  print( training loss[epochs-1], valid loss[epochs-1], test loss[epochs-1],
training acc[epochs-1], valid acc[epochs-1], test acc[epochs-1] )
```

Helper functions used to implement SGD

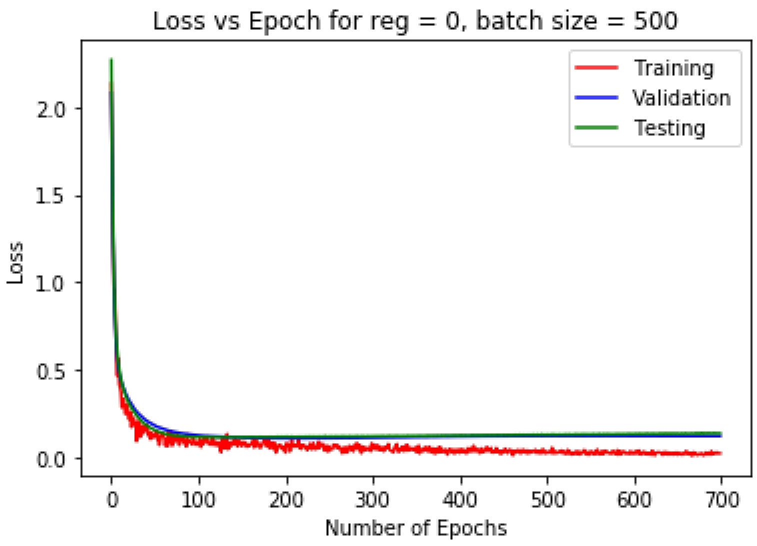
```
def Shuffle( Data, Target, epoch ):
  np.random.seed(epoch)
  randIndx = np.arange( len(Data) )
  np.random.shuffle(randIndx)
  return Data[randIndx], Target[randIndx]

def makeBatch( Data, Target, batchNo, batchSize ):
  lower = batchNo*batchSize
  upper = (batchNo+1)*batchSize
  return Data[lower:upper], Target[lower:upper]
  def Plot( reg, training loss, valid loss, testing loss, epochs, batchSize, training acc,
validation acc, test acc ):

    x axis = np.arange(0, epochs).tolist()

    #plot losses
    title = "Loss vs Epoch for reg = " + str(reg) + ", batch size = " + str(batchSize)
    plt.title(title)
    plt.plot(x axis,training loss,'r', label = "Training")
    plt.plot(x axis,valid loss,'b', label = "Validation")
    plt.plot(x axis,testing loss,'g', label = "Testing")
    plt.legend(loc="upper right")
    plt.ylabel('Loss')
    plt.xlabel('Number of Epochs')
    plt.show()

    #plot accuracies
    title = "Accuracy vs Epoch for reg = " + str(reg) +  ", batch size = " + str(batchSize)
    plt.title(title)
    plt.plot(x axis,training acc,'r', label = "Training")
    plt.plot(x axis,validation acc,'b', label = "Validation")
    plt.plot(x axis,test acc,'g', label = "Testing")
    plt.legend(loc="lower right")
    plt.ylabel('Accuracy')
    plt.xlabel('Number of Epochs')
    plt.show()
```

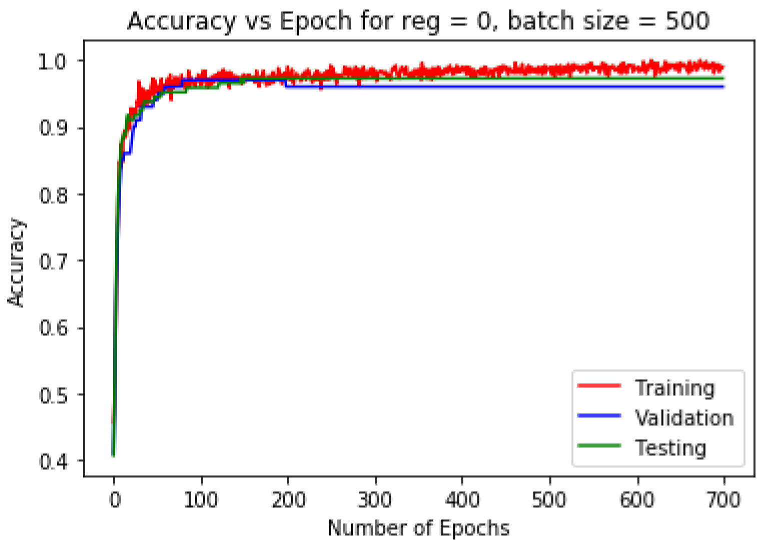
MSE:





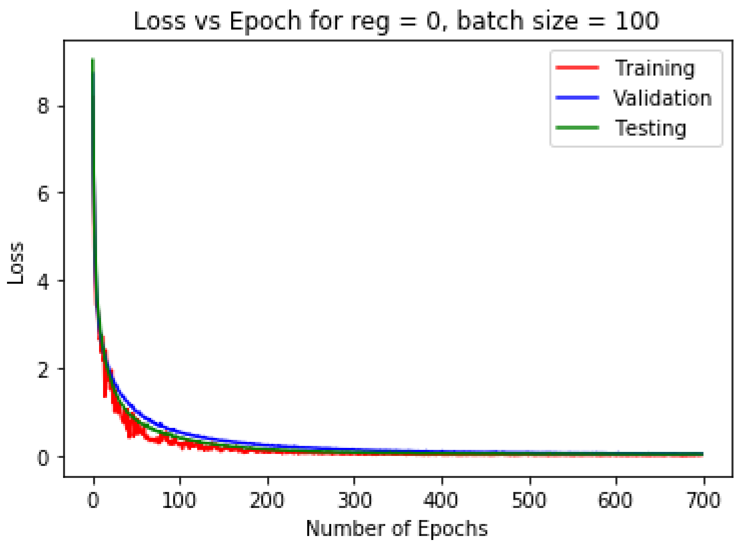
CE:

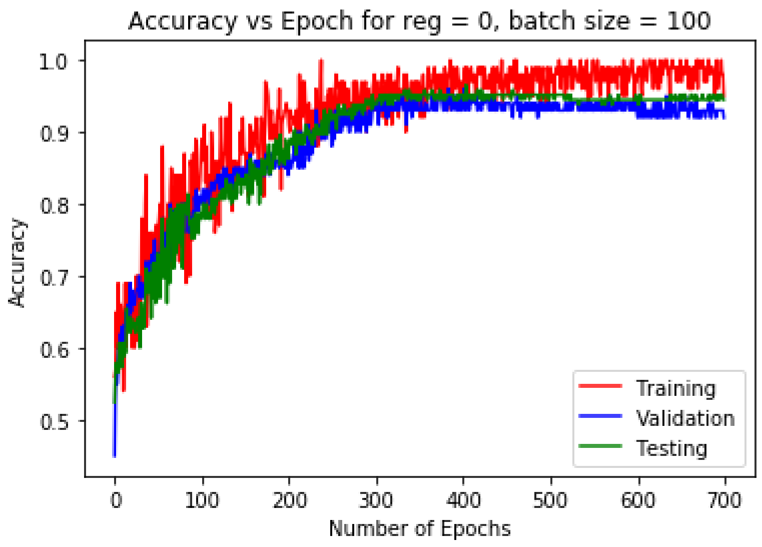


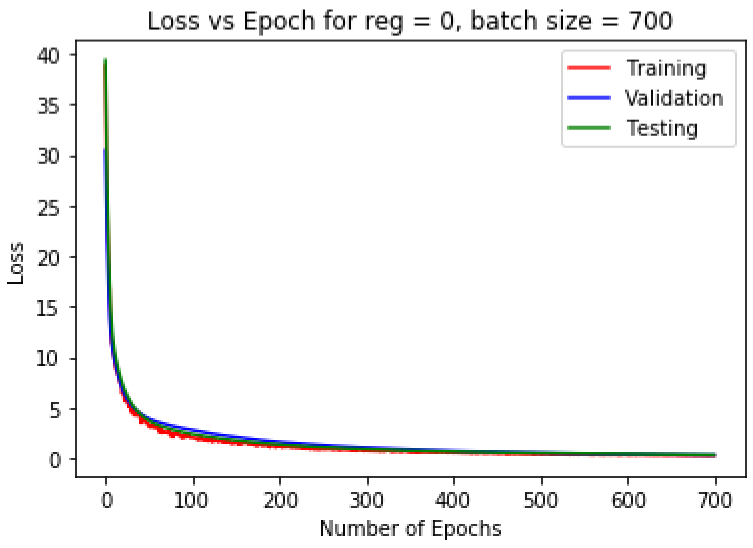


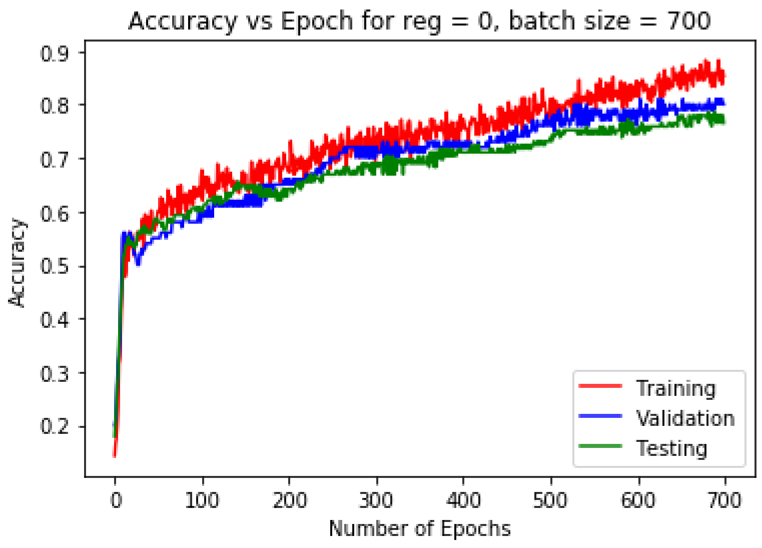
	Final Loss			**Final Accuracy**		
	Training Data	Valid Data	Test Data	Training Data	Valid Data	Test Data
MSE	0.11832772	0.17132485	0.17433365	0.904	0.91	0.86206895
CE	0.025222924	0.12377698	0.13858579	0.99	0.96	0.9724138

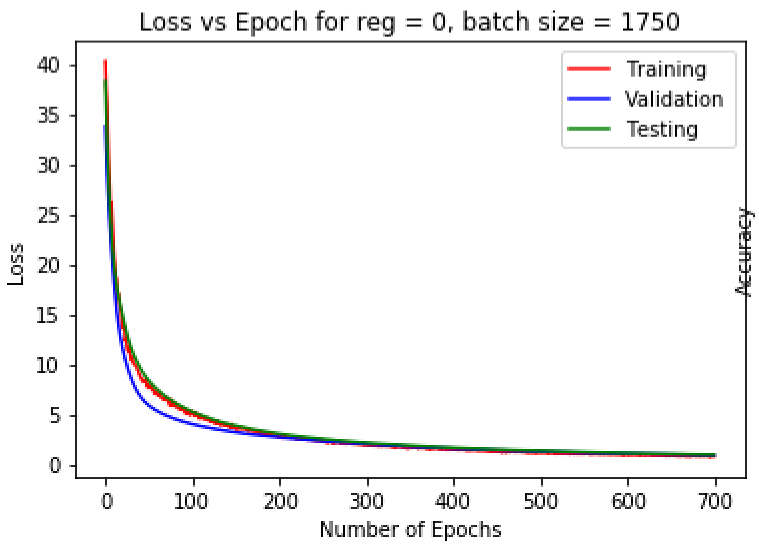
3. Batch Size Investigation

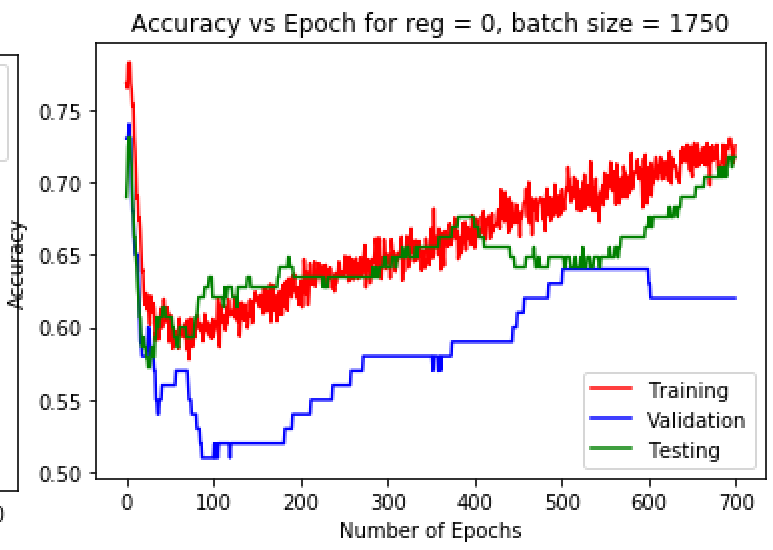












Batch Size	**Final Loss**			**Final Accuracy**		
	Training Data	Valid Data	Test Data	Training Data	Valid Data	Test Data
100	0.03599074	0.062442757	0.05324071	0.95	0.92	0.94482756
700	0.22530618	0.28892514	0.30501333	0.8528572	0.8	0.76551723
1750	0.7921086	0.9146636	0.9538746	0.72514284	0.62	0.7172414

The table above presents final loss and accuracy values from training with batch sizes of 100, 700, and 1750. We can make the observation that as batch sizes increase, loss from training, valid, and test data also increases. As well, the accuracy of our prediction decreases with increased batch size.

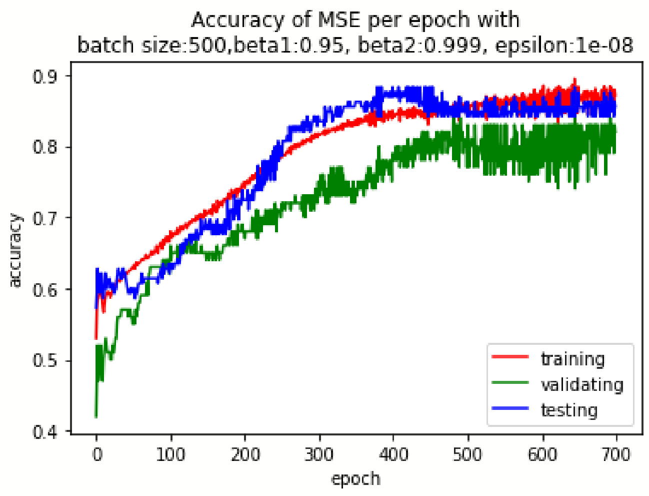
The loss of accuracy with increased batch size could be attributed to the fact that as batch sizes increase, the model updates its weights according to more data points. This could lead to overfitting to the training data, and decreases the model's ability to make generalizations about data it has not been trained against.

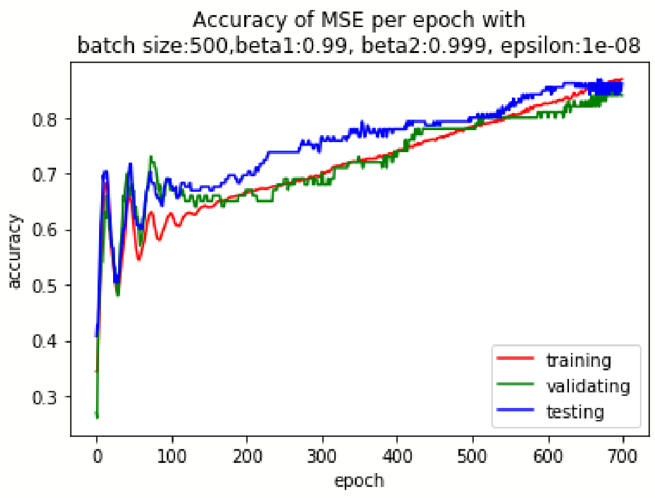
4. Hyperparameter Investigation

MSE

Hyperparameter set $\beta 1$ = [0.95, 0.99]:
Final Accuracy

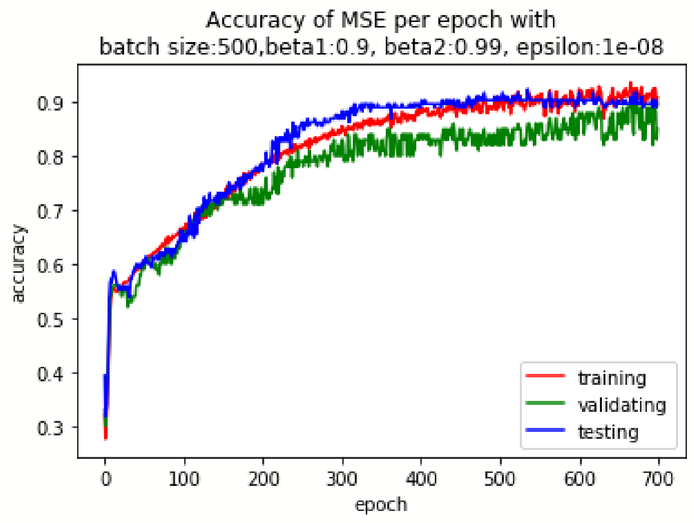
$\beta 1$ = 0.95	-> TrainData: .87,	TestData: .86,	ValidData: 0.82
$\beta 1$ = 0.99	-> TrainData: .87,	TestData: .86,	ValidData: 0.84

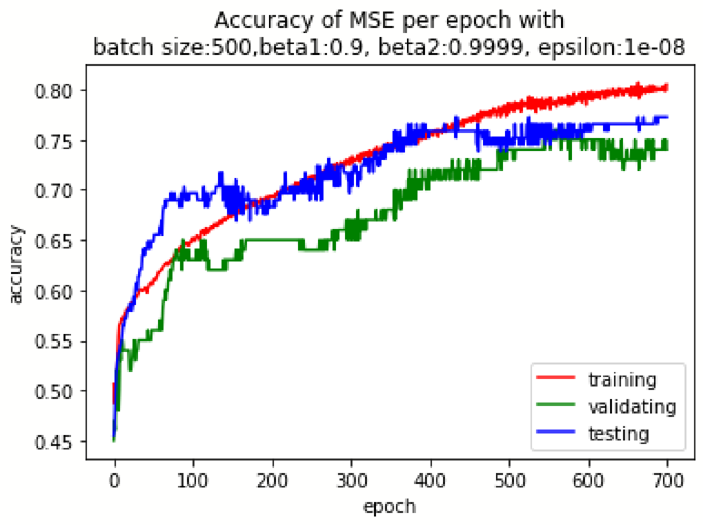




Hyperparameter set $\beta 2$ = [0.99, 0.9999]:
Final Accuracy

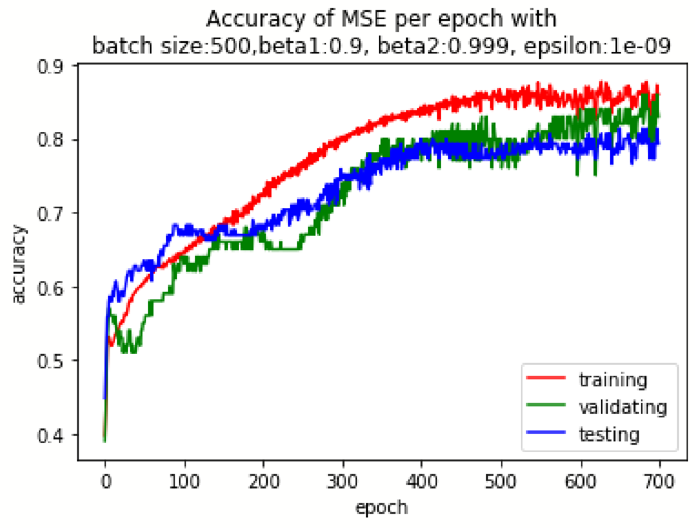
$\beta 2$ = 0.99	->	TrainData: .91,	TestData: .90,	ValidData: .85
$\beta 2$ = 0.9999	->	TrainData: .80,	TestData: .77,	ValidData: .74

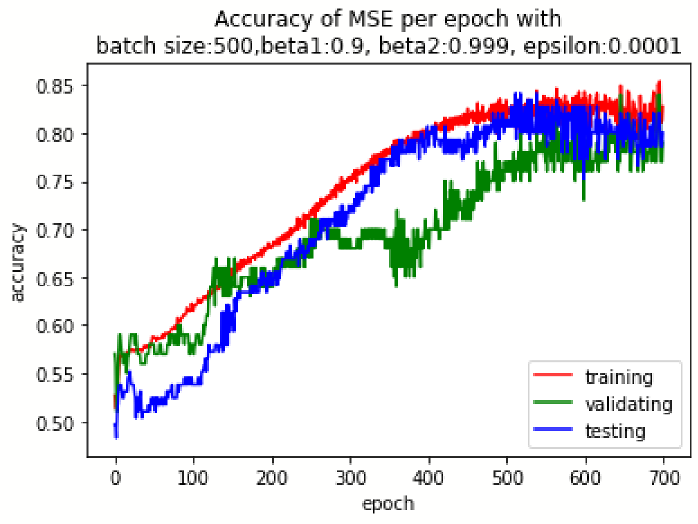




Hyperparameter set ε = [1e-09, 1e-04]:
Final Accuracy

ε = 1e-09	->	TrainData: .86,	TestData: .79,	ValidData: .83
ε = 1e-04	->	TrainData: .83,	TestData: .80,	ValidData: .89





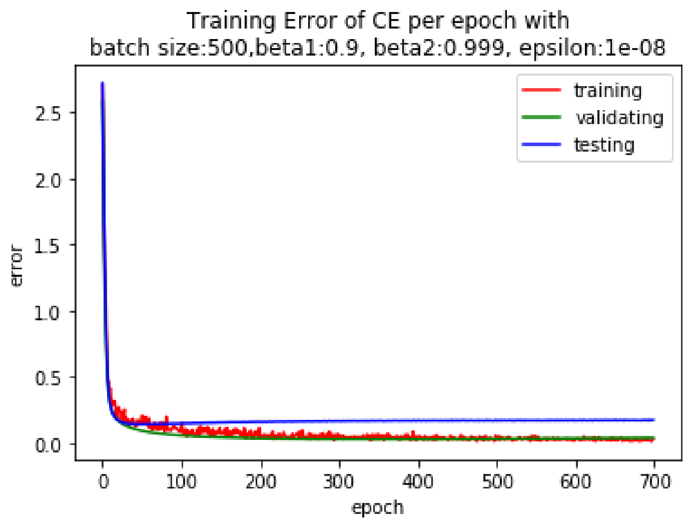
Observing the graphs with varying $\beta 1$ values, we observe that as $\beta 1$ increases, the accuracy graphs becomes smoother, and accuracy of the model prediction slightly improves. Similarly, when $\beta 2$ values were increased from 0.99 to 0.9999, the graph has also become smoother, however accuracy has decreased. According to the documentation, $\beta 1$ determines the first moment used in ADAM, and $\beta 2$ determines the second moment. High β values mean that momentum (historical data) contributes to each weight update.

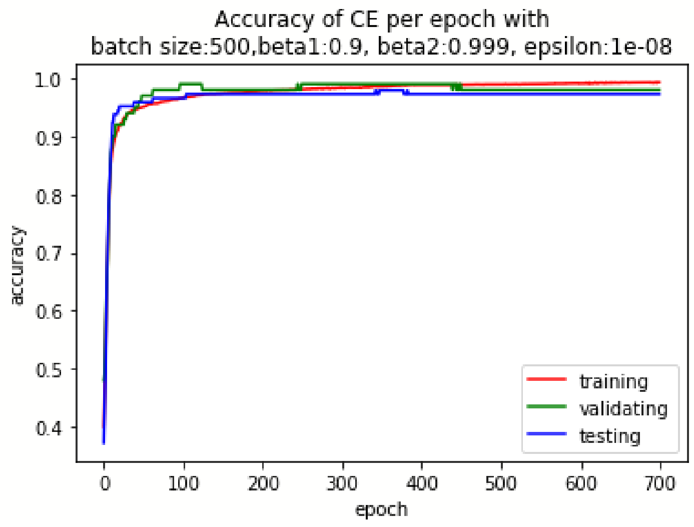
Observing the graphs with varying ε values, we note that as ε is increased, the accuracy of our prediction also increases. The step size (ε) is 100,000 times larger in the second iteration. The higher step size means that in each update of the model's weights, there is a more drastic change in its values.

It is expected that with a higher step size and higher momentum conservation, the model is able to navigate out of local minimums, and find the global minimum for loss faster. We see this happening through the above graphs as we increase each of the 3 hyperparameters.

5. MSE and CE Comparison

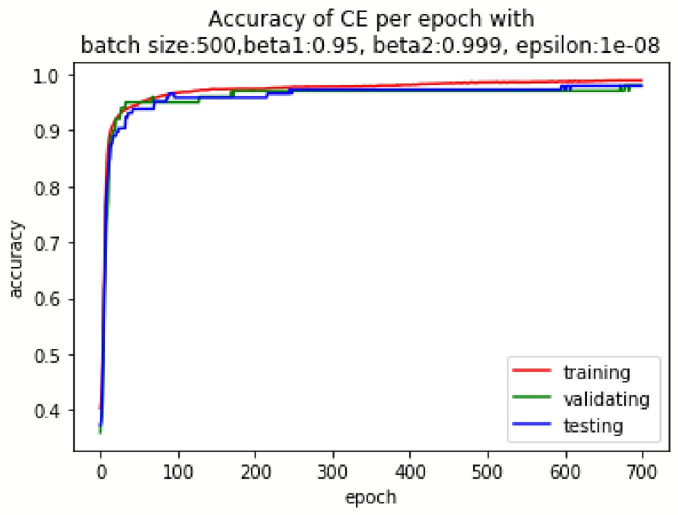
Batch size 500 & 700 epoch & default hyperparameter:

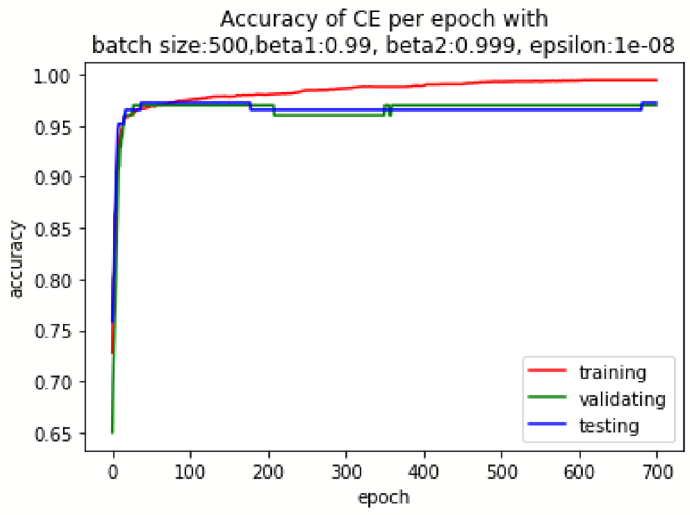




Hyperparameter set $\beta 1$ = [0.95, 0.99]:
Final Accuracy

$\beta 1$ = 0.95 ->	TrainData: .99,	TestData: .98,	ValidData: 0.98
$\beta 1$ = 0.99 ->	TrainData: .99,	TestData: .97,	ValidData: 0.97

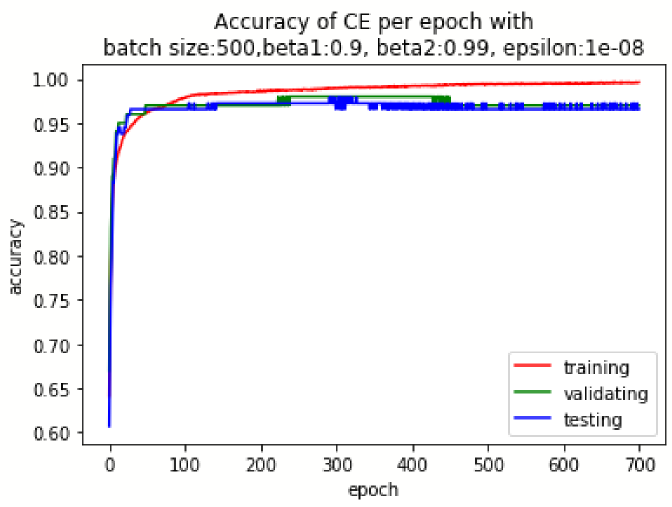


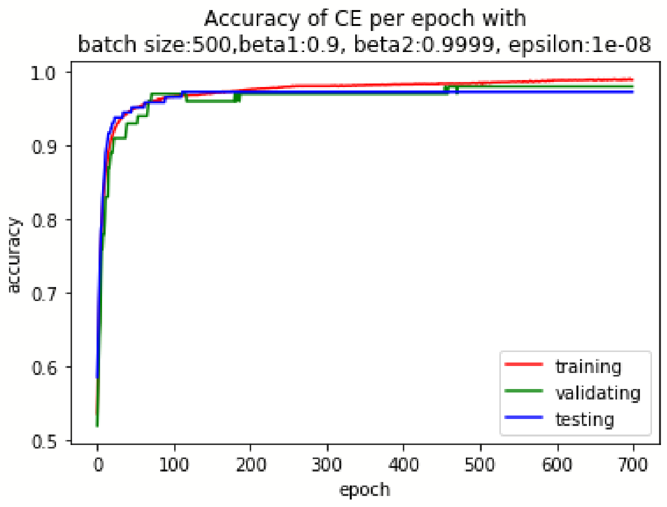


Hyperparameter set $\beta 2$ = [0.99, 0.9999]:
Final Accuracy

$\beta 2$ = 0.99 ->	TrainData: .99,	TestData: .96,	ValidData: .97

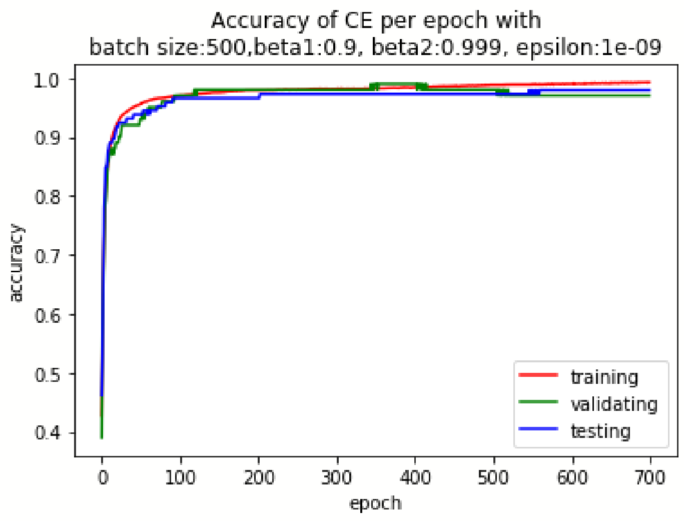
$\beta 2 = 0.9999$ -> TrainData: .99, TestData: .97, ValidData: .96

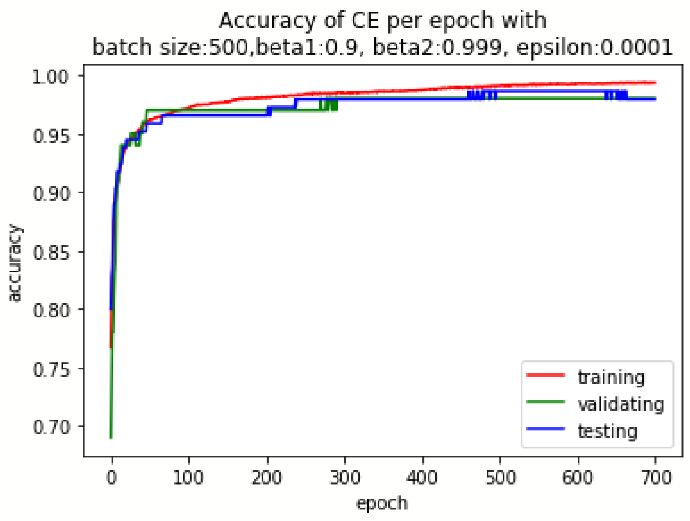




Hyperparameter set ε = [1e-09, 1e-04]:
Final Accuracy

ε = 1e-09	->	TrainData: .99,	TestData: .98,	ValidData: .97
ε = 1e-04	->	TrainData: .99,	TestData: .98,	ValidData: .98





5. MSE and CE Comparison

As observed in the figures in part 3.2, as well as 3.4, using CE results in a faster loss and accuracy convergence. Moreover, the results in the tables of part 3.2 and 3.4 help us conclude that in general, using CE results in a higher accuracy in the model's prediction.

6. ADAM vs Own Implementation

Using ADAM, it is very clear that minimizing CE outperforms minimizing MSE. However, in our own implementation of batch gradient descent, the opposite occurs.

In our own implementation of batch gradient descent, the shapes of the graphs are quite smooth. However, using SGD, the shapes of the graphs are quite jagged. This is because in our own implementation, all data points are passed through, and a single loss and accuracy value is recorded in each epoch. However, in SGD, for all of the batches per epoch that were processed, only the loss and accuracy values of the last batch are graphed. This means that while the loss and accuracy may update slowly over each batch, we only get a value for the last batch processed, making the data points less smooth.