ECE421 – Introduction to Machine Learning

Assignment 1

Linear and Logistic Regression

Shiyi Zhang	Jiani Li
Contribution: 50%	Contribution: 50%

Part I. Linear regression

1. Loss Function and Gradient

- <u>Gradient analytic expression:</u>

  $X\ shape:\ (3500,\ 28*28)$

  $W\ shape:\ (28*28,\ 1)$

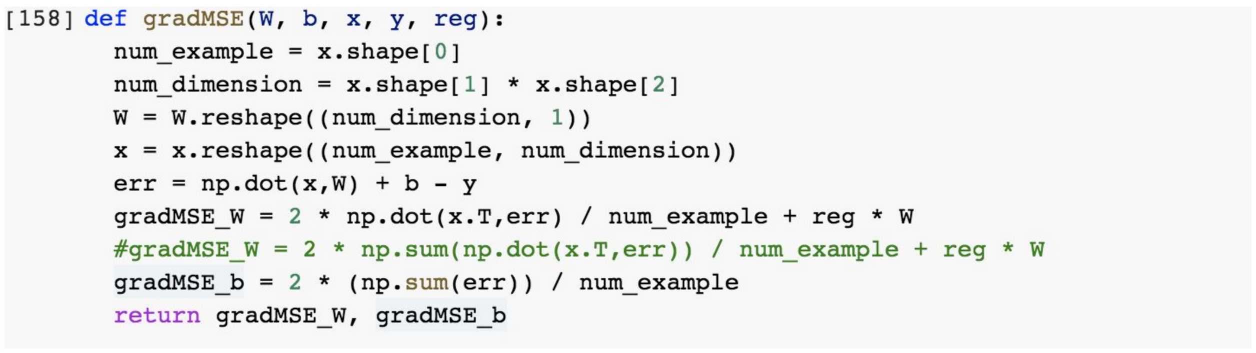
  $y\ shape:\ (3500,\ 1)$

  $$grad_W = \frac{2}{N}(X^T \cdot W + b - y)\ + \lambda \cdot W$$

  $$grad_b = \frac{2}{N}\sum_{n=1}^{N}(X \cdot W + b - y)$$

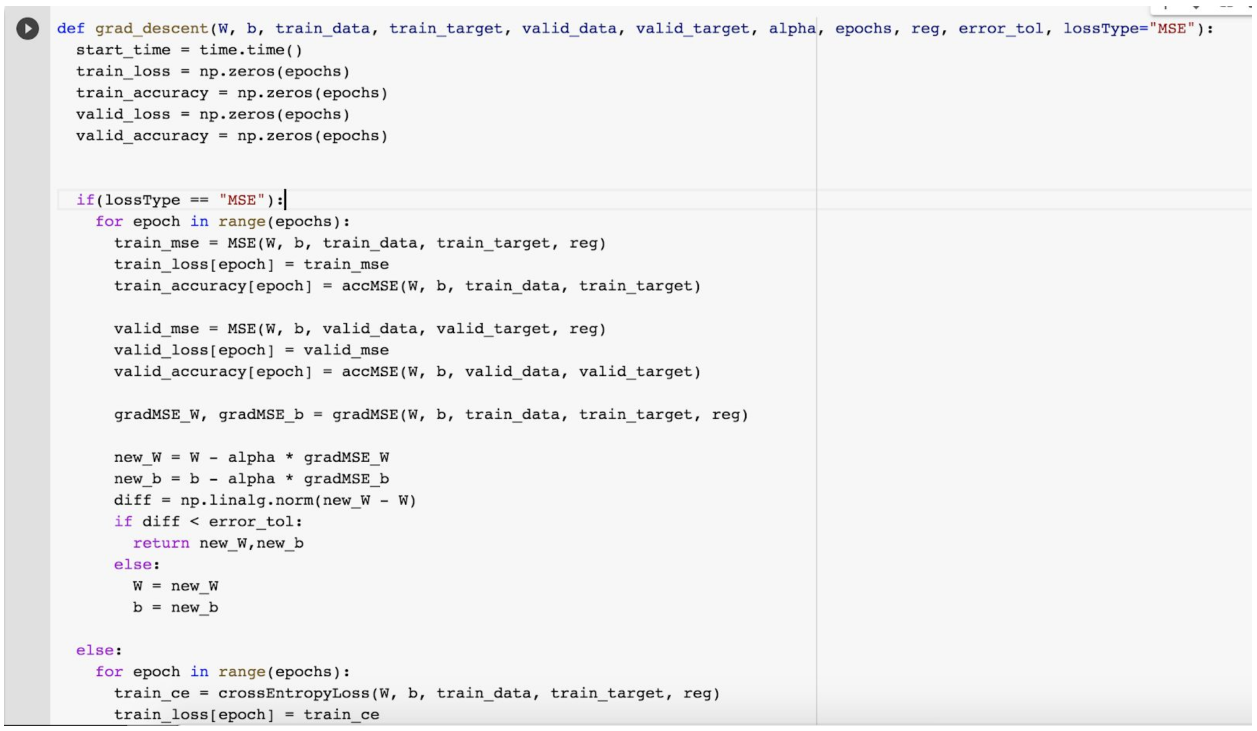
- <u>Code snippet:</u>

```
[137] def MSE(W, b, x, y, reg):
        if len(x.shape)>2:
         num_example = x.shape[0]
         num_dimension = x.shape[1] * x.shape[2]
         W = W.reshape((num_dimension, 1))
         x = x.reshape((num_example, num_dimension))
        err = np.dot(x,W) + b - y
        mse = np.sum(np.dot(err.T,err)) / x.shape[0] + (reg / 2) * np.dot(W.T,W)
        return mse
```



```
[158] def gradMSE(W, b, x, y, reg):
        num_example = x.shape[0]
        num_dimension = x.shape[1] * x.shape[2]
        W = W.reshape((num_dimension, 1))
        x = x.reshape((num_example, num_dimension))
        err = np.dot(x,W) + b - y
        gradMSE_W = 2 * np.dot(x.T,err) / num_example + reg * W
        #gradMSE_W = 2 * np.sum(np.dot(x.T,err)) / num_example + reg * W
        gradMSE_b = 2 * (np.sum(err)) / num_example
        return gradMSE_W, gradMSE_b
```

2. Gradient Descent Implementation



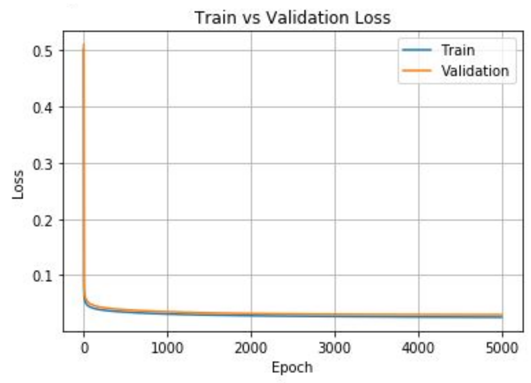
```
def grad_descent(W, b, train_data, train_target, valid_data, valid_target, alpha, epochs, reg, error_tol, lossType="MSE"):
  start_time = time.time()
  train_loss = np.zeros(epochs)
  train_accuracy = np.zeros(epochs)
  valid_loss = np.zeros(epochs)
  valid_accuracy = np.zeros(epochs)


  if(lossType == "MSE"):
    for epoch in range(epochs):
      train_mse = MSE(W, b, train_data, train_target, reg)
      train_loss[epoch] = train_mse
      train_accuracy[epoch] = accMSE(W, b, train_data, train_target)

      valid_mse = MSE(W, b, valid_data, valid_target, reg)
      valid_loss[epoch] = valid_mse
      valid_accuracy[epoch] = accMSE(W, b, valid_data, valid_target)

      gradMSE_W, gradMSE_b = gradMSE(W, b, train_data, train_target, reg)

      new_W = W - alpha * gradMSE_W
      new_b = b - alpha * gradMSE_b
      diff = np.linalg.norm(new_W - W)
      if diff < error_tol:
        return new_W,new_b
      else:
        W = new_W
        b = new_b

  else:
    for epoch in range(epochs):
      train_ce = crossEntropyLoss(W, b, train_data, train_target, reg)
      train_loss[epoch] = train_ce
      train_accuracy[epoch] = accCE(W, b, train_data, train_target)

      valid_ce = crossEntropyLoss(W, b, valid_data, valid_target, reg)
      valid_loss[epoch] = valid_ce
      valid_accuracy[epoch] = accCE(W, b, valid_data, valid_target)

      gradCE_W, gradCE_b = gradCE(W, b, train_data, train_target, reg)

      new_W = W - alpha * gradCE_W
      new_b = b - alpha * gradCE_b
      diff = np.linalg.norm(new_W - W)
      if diff < error_tol:
        return new_W,new_b
      else:
        W = new_W
        b = new_b

  end_time = time.time()
  diff_time = end_time - start_time
  print("GD elapse time: ", diff_time)
  return W, b, train_loss, valid_loss, train_accuracy, valid_accuracy
```

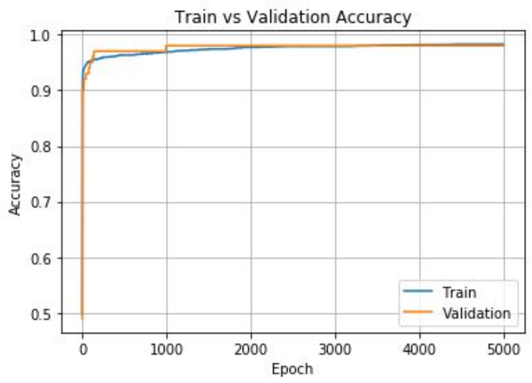
3. Tuning the Learning Rate

Keep epoch = 5000, $\lambda = 0$

Train vs Validation Loss/Accuracy Plots:

1) alpha = 0.005





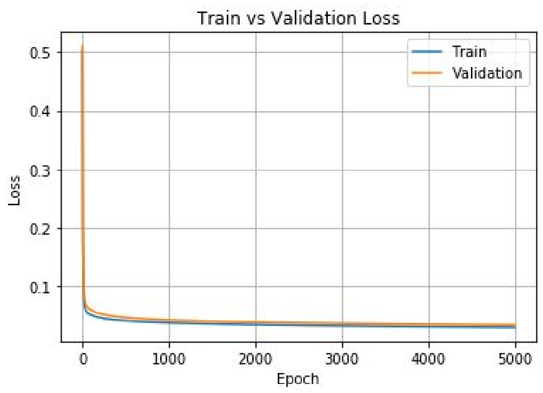
Final train loss: 0.025384655294266374

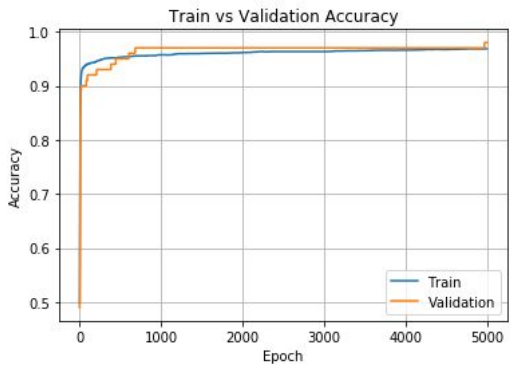
Final train accuracy: 0.9825714285714285

Final validation loss: 0.029890557414703808

Final validation accuracy: 0.98

2) alpha = 0.001





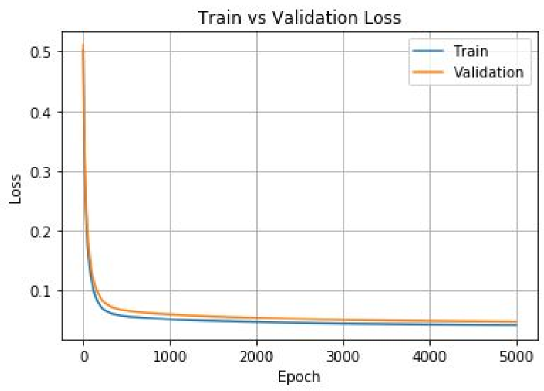
Final train loss: 0.031079269011388952

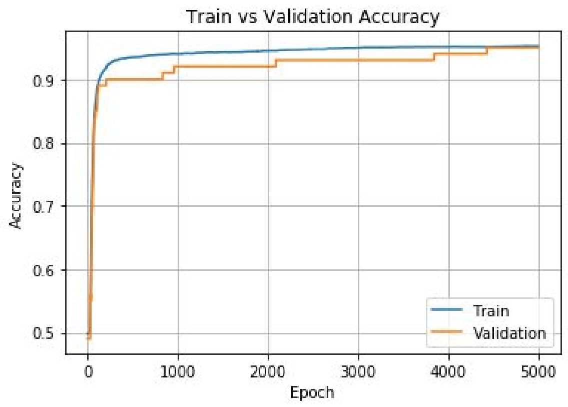
Final train accuracy: 0.9685714285714285

Final validation loss: 0.035013009288847986

Final validation accuracy: 0.98

3) alpha = 0.0001





Final train loss: 0.042093922794912704

Final train accuracy: 0.9525714285714286

Final validation loss: 0.04758644859498016

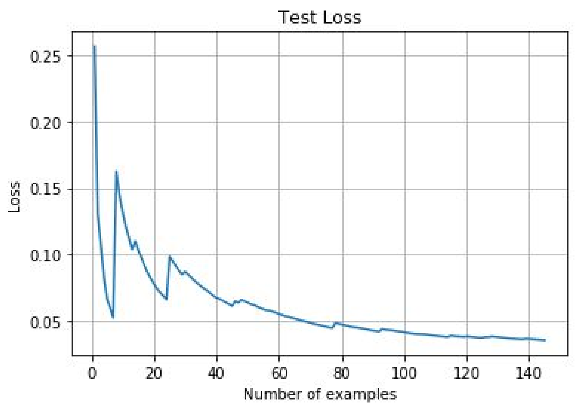
Final validation accuracy: 0.95

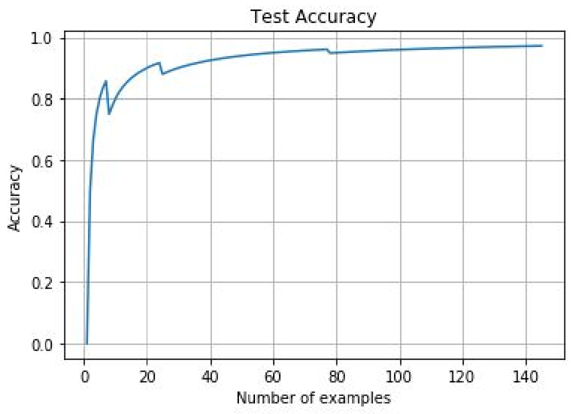
Final training and validation results:

Learning rate	**Train accuracy**	**Validation accuracy**
0.005	0.9825714285714285	0.98
0.001	0.9685714285714285	0.98
0.0001	0.9525714285714286	0.95

Test Loss/Accuracy Plots:

1) alpha = 0.005

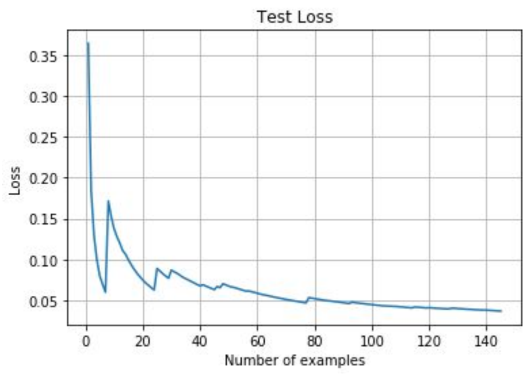


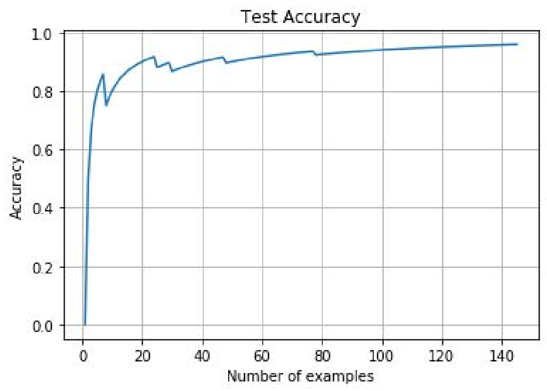


Final test loss: 0.03536328774473713

Final test accuracy: 0.9724137931034482

2) alpha = 0.001

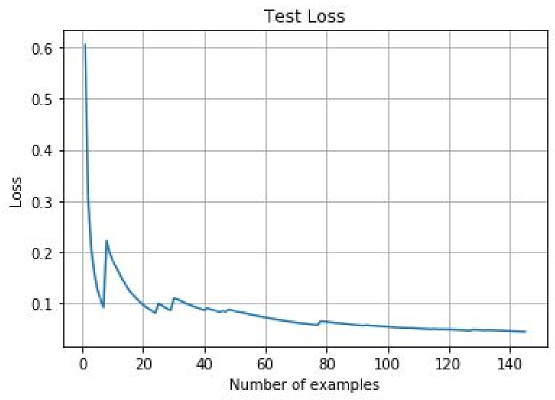


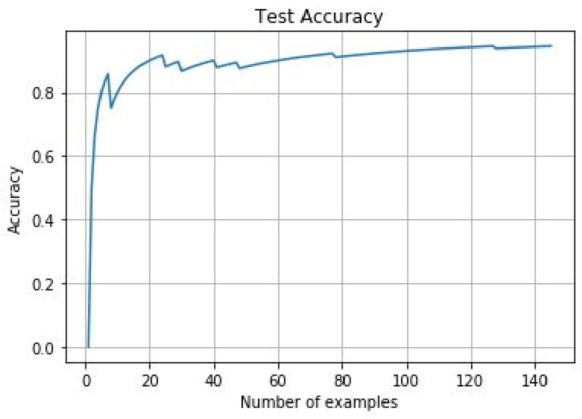


Final test loss: 0.03660526642223174

Final test accuracy: 0.9586206896551724

3) alpha = 0.0001





Final test loss: 0.044859376832925874

Final test accuracy: 0.9448275862068966

Final test results:

Learning rate	**Test loss**	**Test accuracy**
0.005	0.03536328774473713	0.9724137931034482
0.001	0.03660526642223174	0.9586206896551724
0.0001	0.044859376832925874	0.9448275862068966

The test results show that 0.005 is the best learning rate for this model, which has the lowest loss and highest accuracy for both test/validation and test data.

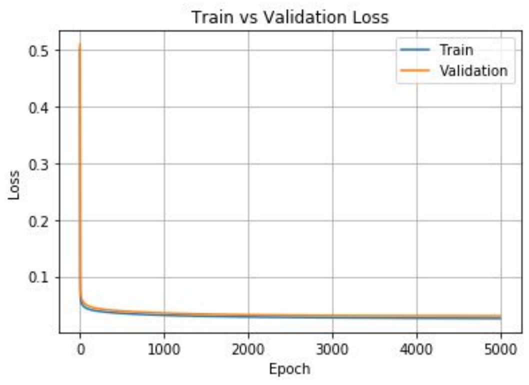
Based on the plots, smaller learning rates require more epochs to converge, therefore it takes longer to train. However, if the learning rate is too large, overshooting may occur.

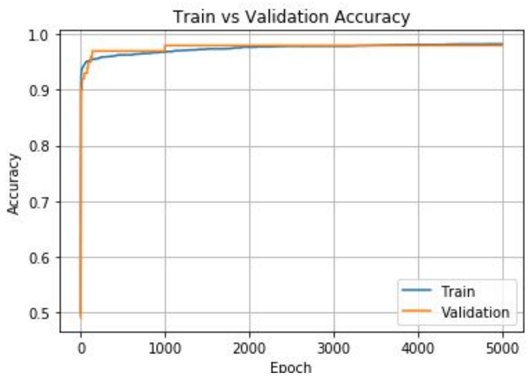
4. Generalization

Keep epoch = 5000, alpha = 0.005

Train vs Validation Loss/Accuracy Plots:

1) λ = 0.001





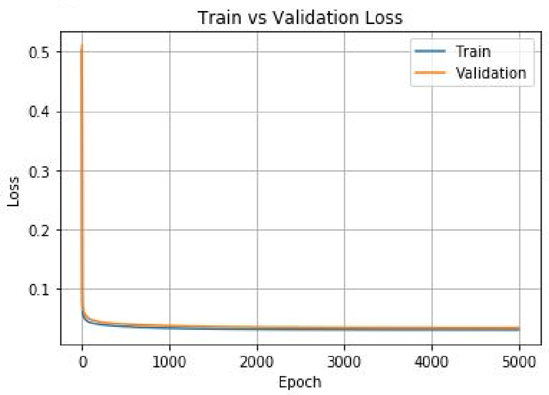
Final train loss: 0.025501921271808958

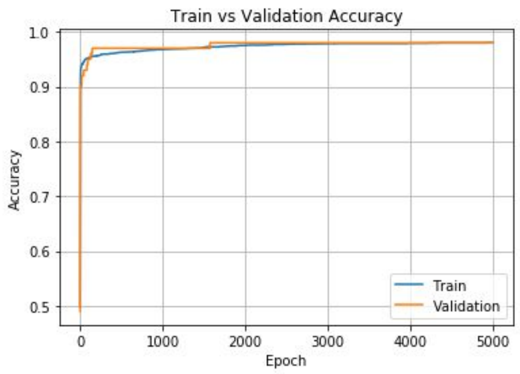
Final train accuracy: 0.9828571428571429

Final validation loss: 0.029982048024671793

Final validation accuracy: 0.98

2) λ = 0.1





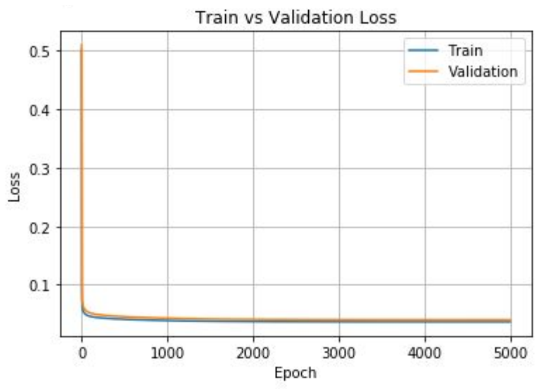
Final train loss: 0.03093960139578801

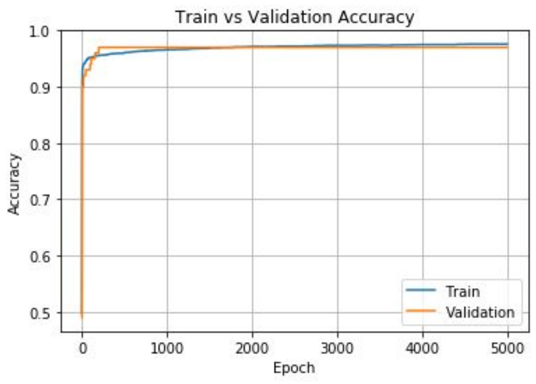
Final train accuracy: 0.9802857142857143

Final validation loss: 0.03426911404502905

Final validation accuracy: 0.98

3) λ = 0.5





Final train loss: 0.03674159281413293

Final train accuracy: 0.9757142857142858

Final validation loss: 0.04019693041267704

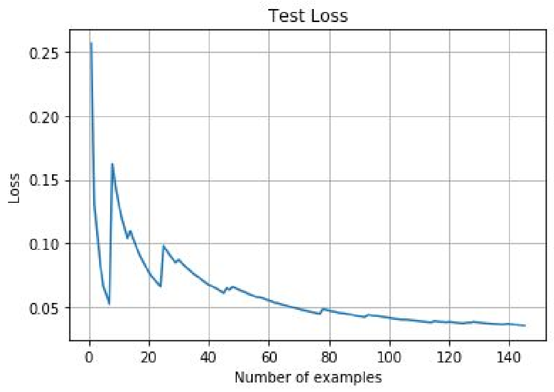
Final validation accuracy: 0.97

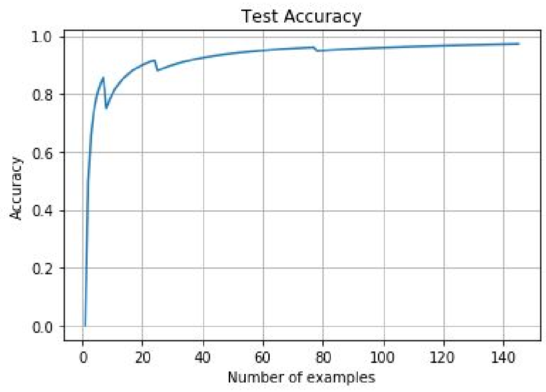
<u>Final training and validation results:</u>

Regularization Parameter	**Train accuracy**	**Validation accuracy**
0.001	0.9828571428571429	0.98
0.1	0.9802857142857143	0.98
0.5	0.9757142857142858	0.97

Test Loss/Accuracy Plots:

1) λ = 0.001

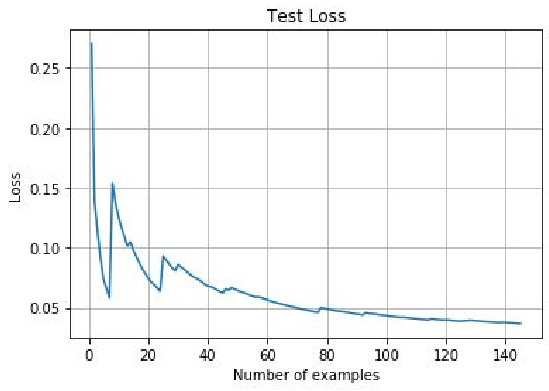


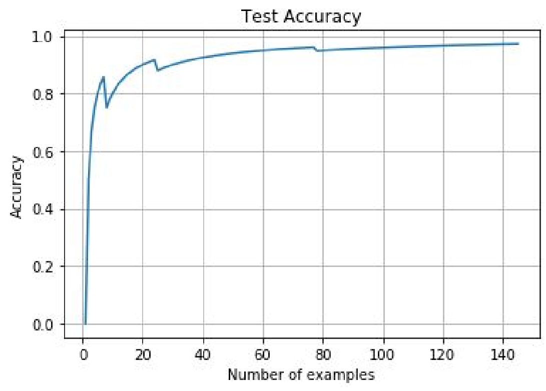


Final test loss: 0.03540431112847912

Final test accuracy: 0.9724137931034482

2) λ = 0.1

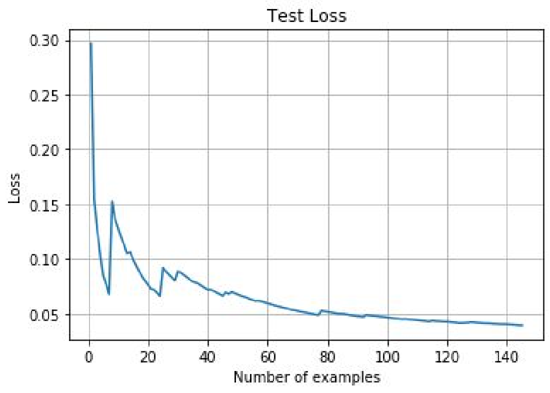


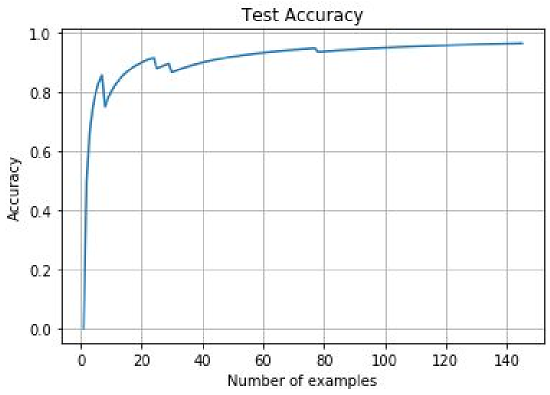


Final test loss: 0.036572751091303424

Final test accuracy: 0.9724137931034482

3) λ = 0.5





Final test loss: 0.039333158742316035

Final test accuracy: 0.9655172413793104

<u>Test results:</u>

Regularization Parameter	**Test loss**	**Test accuracy**
0.001	0.03540431112847912	0.9724137931034482
0.1	0.036572751091303424	0.9724137931034482
0.5	0.039333158742316035	0.9655172413793104

As indicated in the tables, a regularization of 0.001 achieved the smallest test loss and highest train, validation and test accuracy, thus it is the best regularization parameter among the three for this model.

Based on the plots, smaller regularization parameters require more epochs to converge. As the magnitudes of the regularization parameter increase, there will be an increasing penalty on the cost function, which contributes more to avoiding the occurrence of overfitting.

5. Comparing Batch GD with Normal Equation

Method	**Final train MSE**	**Final train accuracy**	**Time**
Normal equation	0.023164031854773937	0.9842857142857143	0.3065025806427002
Batch GD	0.025384655294266374	0.9825714285714285	28.19485592842102

In terms of MSE and accuracy, the normal equation has a better performance since it is the optimal solution for the training set. But for test results, Batch GD should be better because normal equation may get over-fitting. Normal equation takes less time than Batch GD because though the inverse matrix of a (28*28, 28*28) takes time to compute, Batch GD goes through many iterations (epochs), which is 5000 in this case and does matrix calculations many times. Therefore, it makes sense that to deal with this problem, the Normal equation method takes shorter.

However, as the size of the input data matrix increases, normal equation computation can become far more time-consuming. Therefore, to avoid large amounts of matrix computation and the occurrence of over-fitting, Batch GD is a better way to go, and we can always choose a reasonable number of training epochs to avoid overfitting.

Part II. Logistic regression

1. Loss function and gradient

- Gradient analytic expression:

$X\ shape:\ (3500,\ 28 * 28)$

$W\ shape:\ (28 * 28,\ 1)$

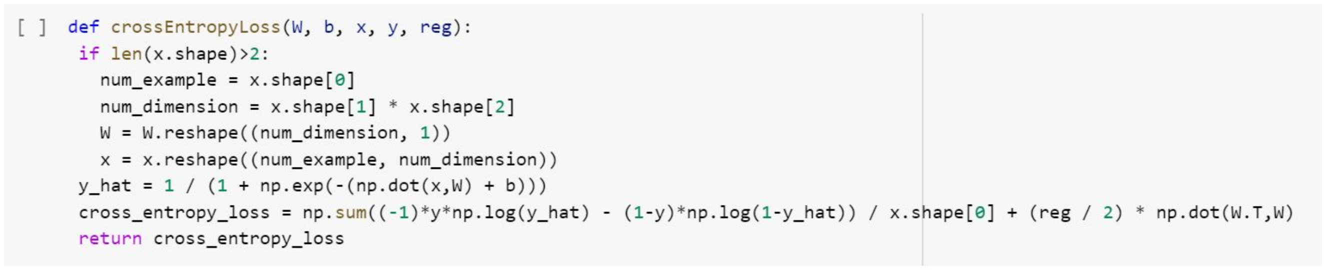
$y\ shape:\ (3500,\ 1)$

$$grad_W = \frac{2}{N}(X^T \cdot (\hat{y} - y)) + \lambda \cdot W$$

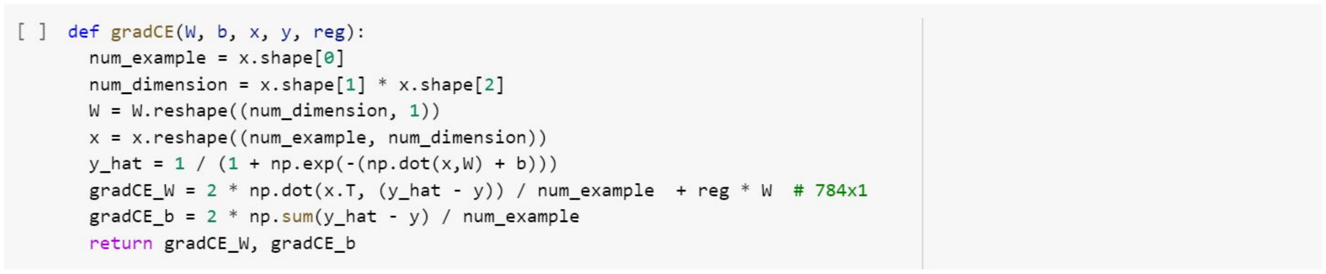
$$grad_b = \frac{2}{N} \sum_{n=1}^{N} (\hat{y} - y)$$

where $\hat{y} = \frac{1}{1 + e^{-(x \cdot W + b)}}$

- Code snippet:



```
[ ]  def crossEntropyLoss(W, b, x, y, reg):
       if len(x.shape)>2:
         num_example = x.shape[0]
         num_dimension = x.shape[1] * x.shape[2]
         W = W.reshape((num_dimension, 1))
         x = x.reshape((num_example, num_dimension))
       y_hat = 1 / (1 + np.exp(-(np.dot(x,W) + b)))
       cross_entropy_loss = np.sum((-1)*y*np.log(y_hat) - (1-y)*np.log(1-y_hat)) / x.shape[0] + (reg / 2) * np.dot(W.T,W)
       return cross_entropy_loss
```

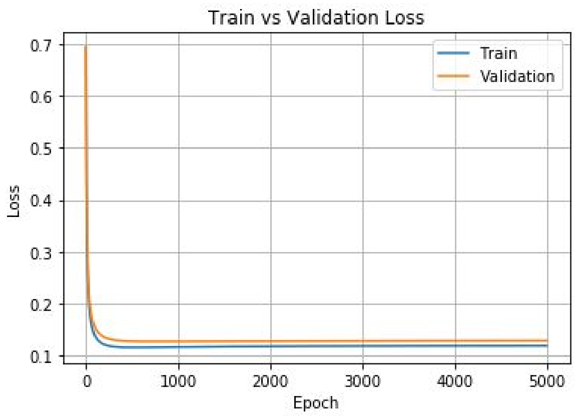


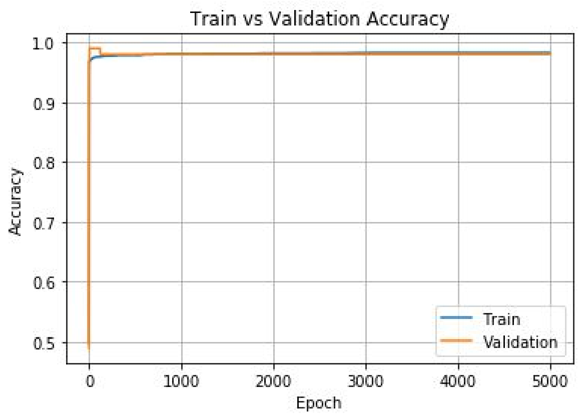
```
[ ]  def gradCE(W, b, x, y, reg):
        num_example = x.shape[0]
        num_dimension = x.shape[1] * x.shape[2]
        W = W.reshape((num_dimension, 1))
        x = x.reshape((num_example, num_dimension))
        y_hat = 1 / (1 + np.exp(-(np.dot(x,W) + b)))
        gradCE_W = 2 * np.dot(x.T, (y_hat - y)) / num_example  + reg * W  # 784x1
        gradCE_b = 2 * np.sum(y_hat - y) / num_example
        return gradCE_W, gradCE_b
```

2. Learning

Set $\alpha = 0.005$, epoch = 5000, $\lambda = 0.1$

Train vs Validation Loss/Accuracy Plots:





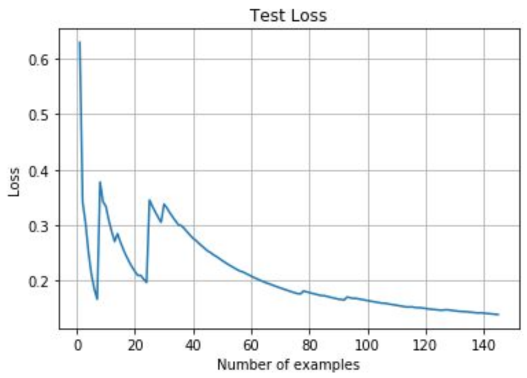
Final train loss: 0.11933795174595027

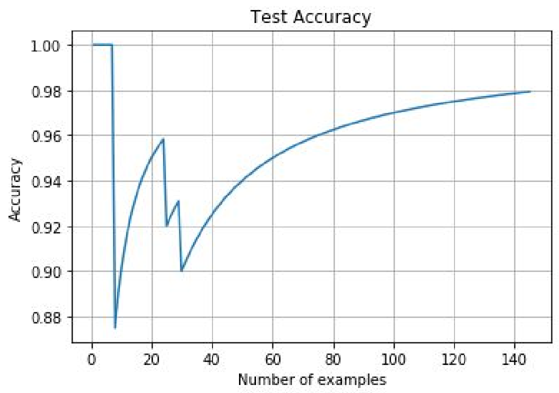
Final train accuracy: 0.9828571428571429

Final validation loss: 0.12923377230889446

Final validation accuracy: 0.98

Test Loss/Accuracy Plots:





Final test loss: 0.13908045770311783

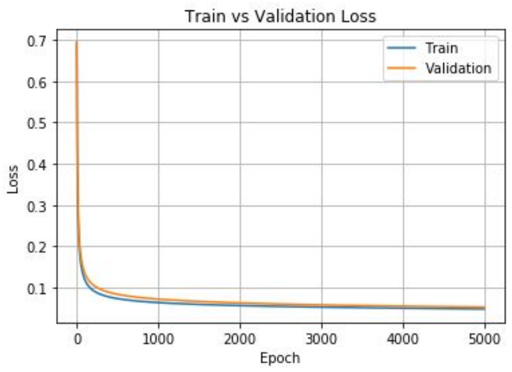
Final test accuracy: 0.9793103448275862

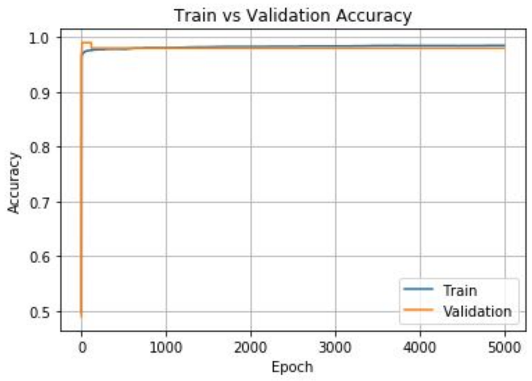
3. Comparison to Linear Regression

Set $\alpha = 0.005$, epoch = 5000, $\lambda = 0$

1) Cross Entropy Loss

<u>Train vs Validation Loss/Accuracy Plots:</u>





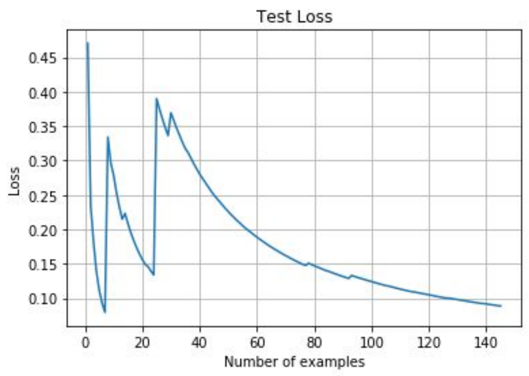
Final train loss: 0.0481598363021075

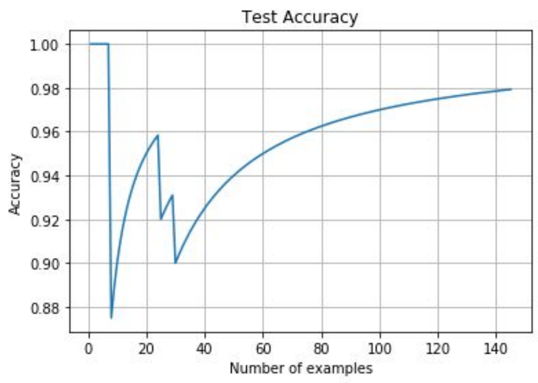
Final train accuracy: 0.98485714285714292

Final validation loss: 0.05280032579646205

Final validation accuracy: 0.98

<u>Test Loss/Accuracy Plots:</u>



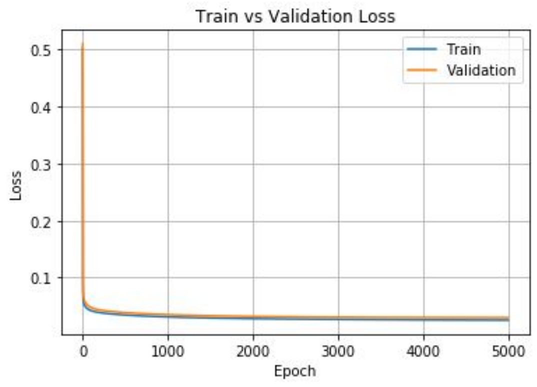


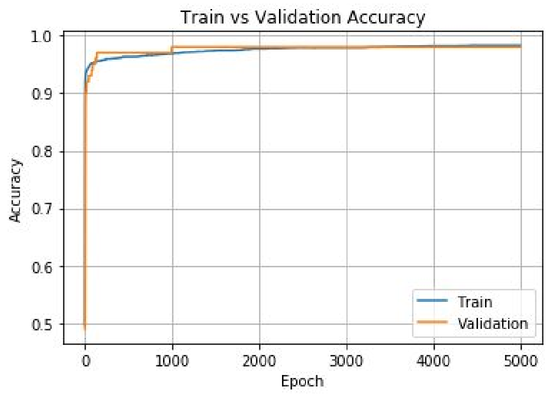
Final test loss: 0.08846602141763323

Final test accuracy: 0.9793103448275862

2) MSE

Train vs Validation Loss/Accuracy Plots:





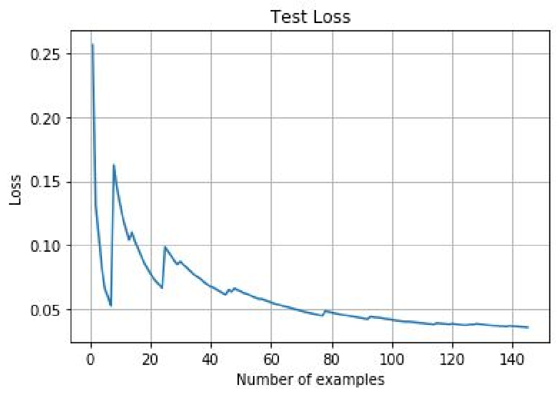
Final train loss: 0.025384655294266374

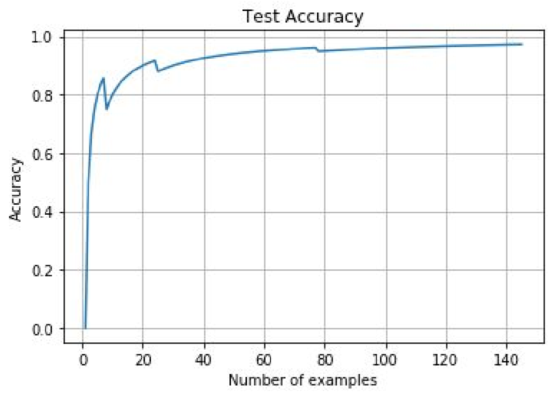
Final train accuracy: 0.9825714285714285

Final validation loss: 0.029890557414703808

Final validation accuracy: 0.98

Test Loss/Accuracy Plots:





Final test loss: 0.035363287744473713

Final test accuracy: 0.9724137931034482

Comparison:

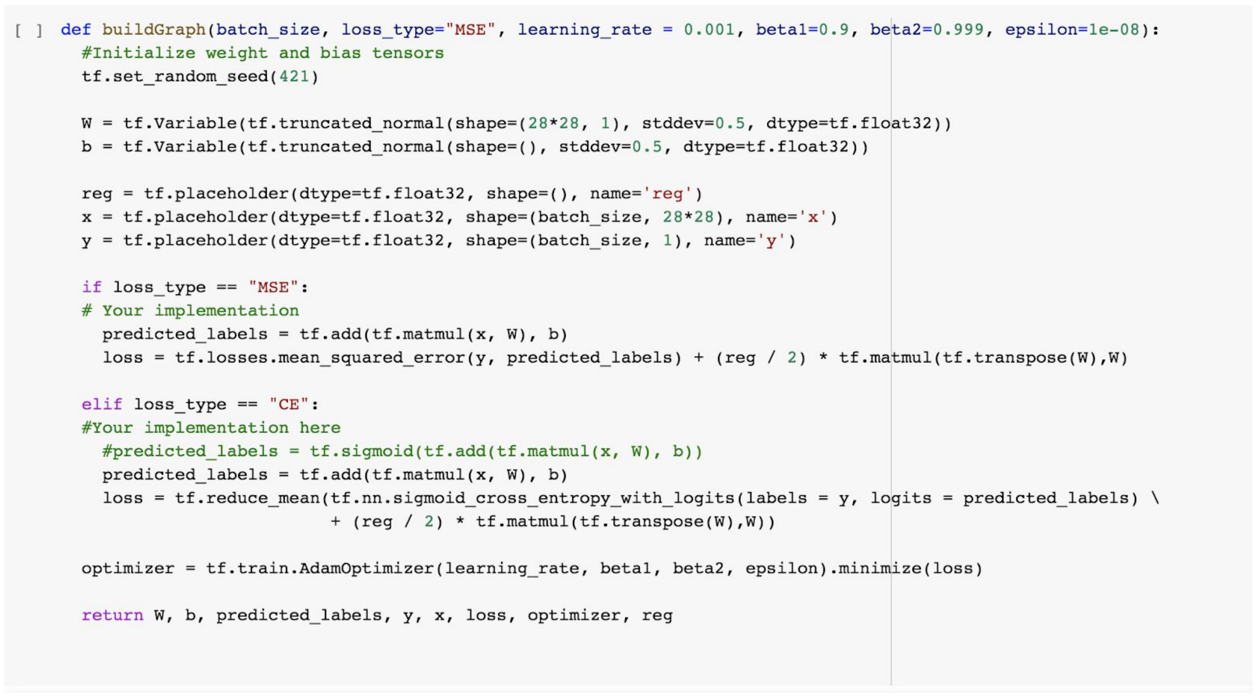
Loss Type	**Test Loss**	**Test Accuracy**

Cross Entropy Loss	0.08846602141763323	0.9793103448275862
MSE	0.03536328774473713	0.9724137931034482

Based on the plots, cross-entropy loss enables the model to converge faster. The effect of cross-entropy loss convergence behaviour can help avoid the occurrence of a slowdown or even full stagnation. For MSE, many misclassified training examples can result in a tiny gradient and may lead to the vanishing gradient problem. While for cross entropy loss, the term which gets smaller and smaller in MSE goes away, this has caused a higher loss, but with this kind of convergence effect, the training is unlikely to stall out.

Part III. Batch gradient descent VS SGD and Adam

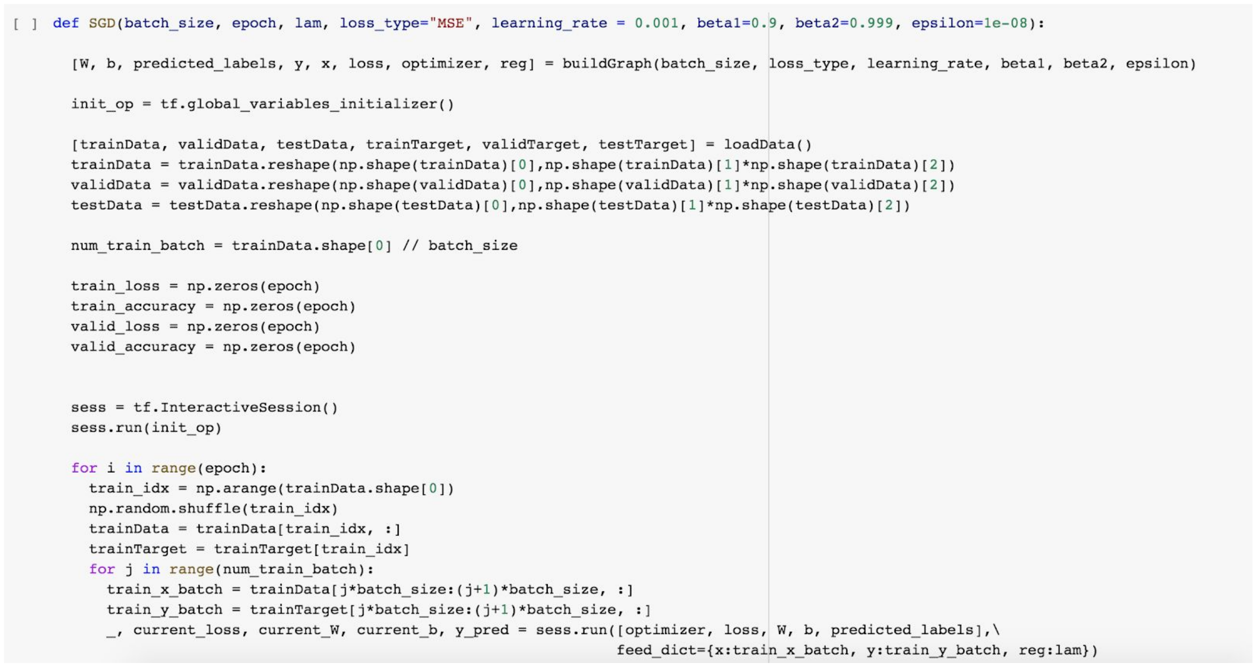
1. Building the Computational Graph



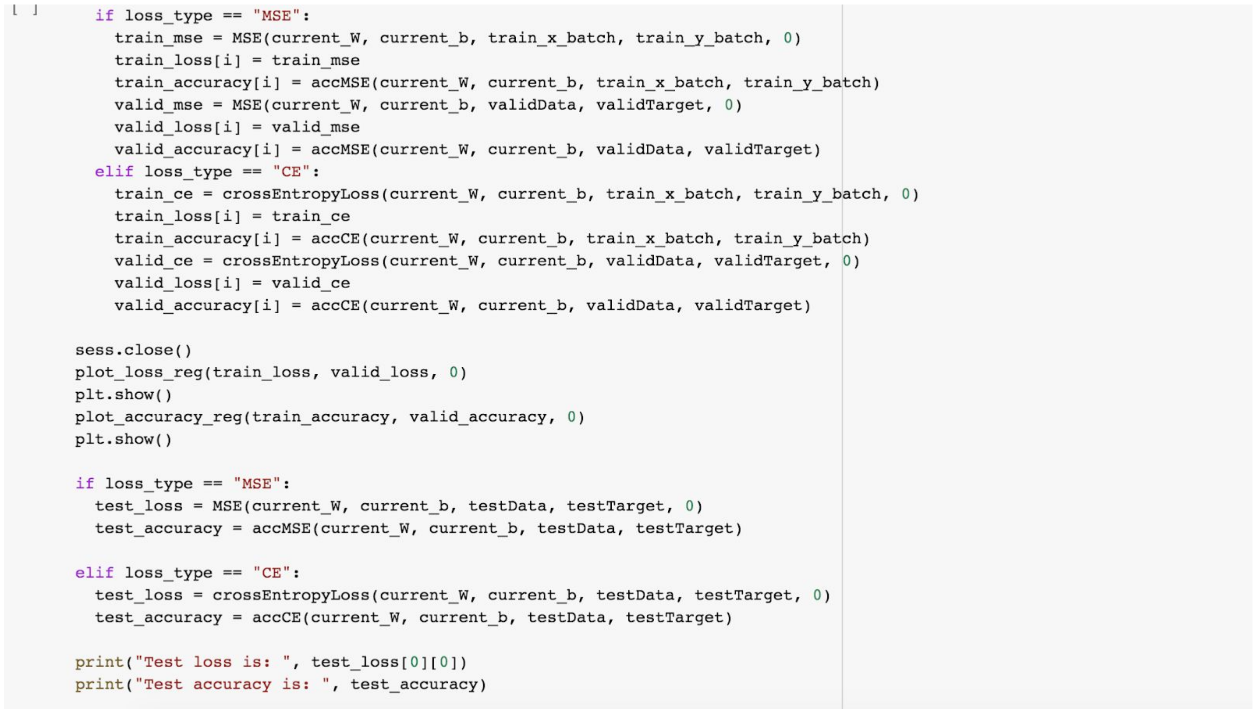
```
[ ] def buildGraph(batch_size, loss_type="MSE", learning_rate = 0.001, beta1=0.9, beta2=0.999, epsilon=1e-08):
      #Initialize weight and bias tensors
      tf.set_random_seed(421)

      W = tf.Variable(tf.truncated_normal(shape=(28*28, 1), stddev=0.5, dtype=tf.float32))
      b = tf.Variable(tf.truncated_normal(shape=(), stddev=0.5, dtype=tf.float32))

      reg = tf.placeholder(dtype=tf.float32, shape=(), name='reg')
      x = tf.placeholder(dtype=tf.float32, shape=(batch_size, 28*28), name='x')
      y = tf.placeholder(dtype=tf.float32, shape=(batch_size, 1), name='y')

      if loss_type == "MSE":
      # Your implementation
        predicted_labels = tf.add(tf.matmul(x, W), b)
        loss = tf.losses.mean_squared_error(y, predicted_labels) + (reg / 2) * tf.matmul(tf.transpose(W),W)

      elif loss_type == "CE":
      #Your implementation here
        #predicted_labels = tf.sigmoid(tf.add(tf.matmul(x, W), b))
        predicted_labels = tf.add(tf.matmul(x, W), b)
        loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels = y, logits = predicted_labels) \
                            + (reg / 2) * tf.matmul(tf.transpose(W),W))

      optimizer = tf.train.AdamOptimizer(learning_rate, beta1, beta2, epsilon).minimize(loss)

      return W, b, predicted_labels, y, x, loss, optimizer, reg
```

2. Implementing Stochastic Gradient Descent

- <u>Code snippet:</u>



```
[ ] def SGD(batch_size, epoch, lam, loss_type="MSE", learning_rate = 0.001, beta1=0.9, beta2=0.999, epsilon=1e-08):

      [W, b, predicted_labels, y, x, loss, optimizer, reg] = buildGraph(batch_size, loss_type, learning_rate, beta1, beta2, epsilon)

      init_op = tf.global_variables_initializer()

      [trainData, validData, testData, trainTarget, validTarget, testTarget] = loadData()
      trainData = trainData.reshape(np.shape(trainData)[0],np.shape(trainData)[1]*np.shape(trainData)[2])
      validData = validData.reshape(np.shape(validData)[0],np.shape(validData)[1]*np.shape(validData)[2])
      testData = testData.reshape(np.shape(testData)[0],np.shape(testData)[1]*np.shape(testData)[2])

      num_train_batch = trainData.shape[0] // batch_size

      train_loss = np.zeros(epoch)
      train_accuracy = np.zeros(epoch)
      valid_loss = np.zeros(epoch)
      valid_accuracy = np.zeros(epoch)


      sess = tf.InteractiveSession()
      sess.run(init_op)

      for i in range(epoch):
        train_idx = np.arange(trainData.shape[0])
        np.random.shuffle(train_idx)
        trainData = trainData[train_idx, :]
        trainTarget = trainTarget[train_idx]
        for j in range(num_train_batch):
          train_x_batch = trainData[j*batch_size:(j+1)*batch_size, :]
          train_y_batch = trainTarget[j*batch_size:(j+1)*batch_size, :]
          _, current_loss, current_W, current_b, y_pred = sess.run([optimizer, loss, W, b, predicted_labels],\
                                                                  feed_dict={x:train_x_batch, y:train_y_batch, reg:lam})
```

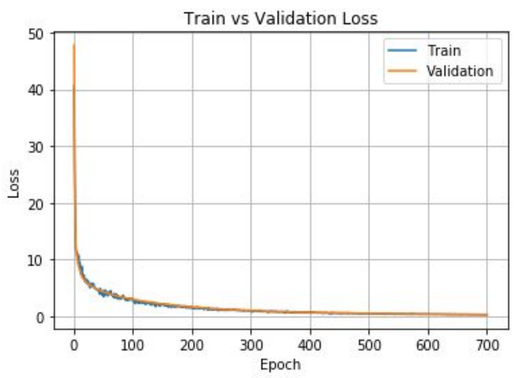


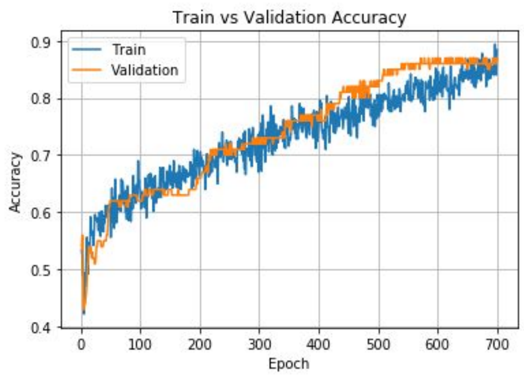
```
[ ]     if loss_type == "MSE":
          train_mse = MSE(current_W, current_b, train_x_batch, train_y_batch, 0)
          train_loss[i] = train_mse
          train_accuracy[i] = accMSE(current_W, current_b, train_x_batch, train_y_batch)
          valid_mse = MSE(current_W, current_b, validData, validTarget, 0)
          valid_loss[i] = valid_mse
          valid_accuracy[i] = accMSE(current_W, current_b, validData, validTarget)
        elif loss_type == "CE":
          train_ce = crossEntropyLoss(current_W, current_b, train_x_batch, train_y_batch, 0)
          train_loss[i] = train_ce
          train_accuracy[i] = accCE(current_W, current_b, train_x_batch, train_y_batch)
          valid_ce = crossEntropyLoss(current_W, current_b, validData, validTarget, 0)
          valid_loss[i] = valid_ce
          valid_accuracy[i] = accCE(current_W, current_b, validData, validTarget)

      sess.close()
      plot_loss_reg(train_loss, valid_loss, 0)
      plt.show()
      plot_accuracy_reg(train_accuracy, valid_accuracy, 0)
      plt.show()

      if loss_type == "MSE":
        test_loss = MSE(current_W, current_b, testData, testTarget, 0)
        test_accuracy = accMSE(current_W, current_b, testData, testTarget)

      elif loss_type == "CE":
        test_loss = crossEntropyLoss(current_W, current_b, testData, testTarget, 0)
        test_accuracy = accCE(current_W, current_b, testData, testTarget)

      print("Test loss is: ", test_loss[0][0])
      print("Test accuracy is: ", test_accuracy)
```

Run SGD with batch_size = 500, epochs = 700 and other parameters = default

```
[ ] SGD(500, 700, 0, loss_type="MSE", learning_rate = 0.001, beta1=0.9, beta2=0.999, epsilon=1e-08)
```

Train vs Validation Loss/Accuracy Plots:





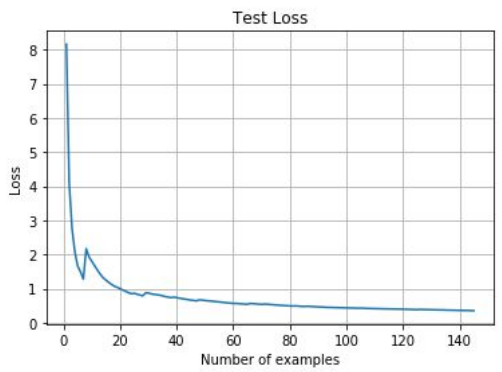
Final train loss: 0.1989019364118576

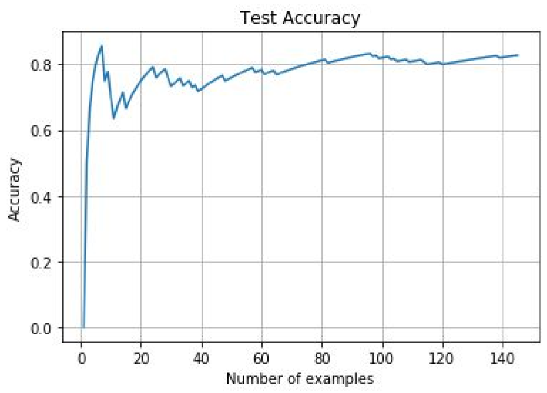
Final train accuracy: 0.884

Final validation loss: 0.27789565920829773

Final validation accuracy: 0.87

Test Loss/Accuracy Plots:





Final test loss: 0.3542379081912838

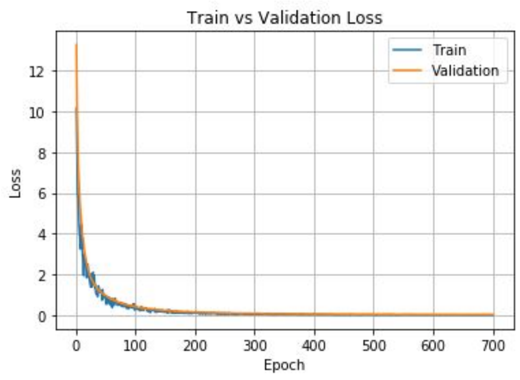
Final test accuracy: 0.8275862068965517

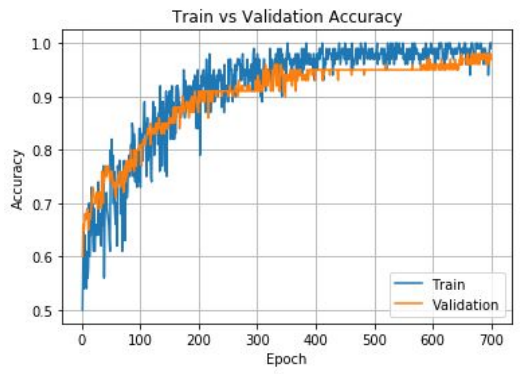
3. Batch Size Investigation

1) Batch size = 100

```
[ ] SGD(100, 700, 0, loss_type="MSE", learning_rate = 0.001, beta1=0.9, beta2=0.999, epsilon=1e-08)
```

<u>Train vs Validation Loss/Accuracy Plots:</u>





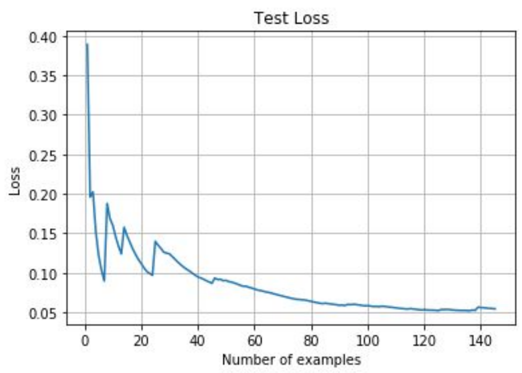
Final train loss: 0.01660042069852352

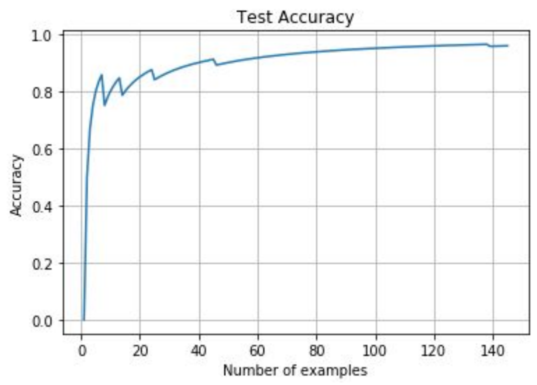
Final train accuracy: 1.0

Final validation loss: 0.047743599861860275

Final validation accuracy: 0.97

<u>Test Loss/Accuracy Plots:</u>





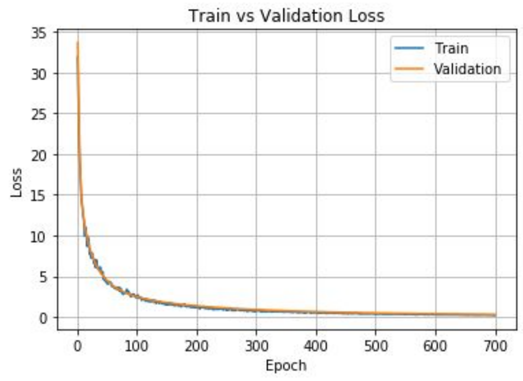
Final test loss: 0.054499444550233526

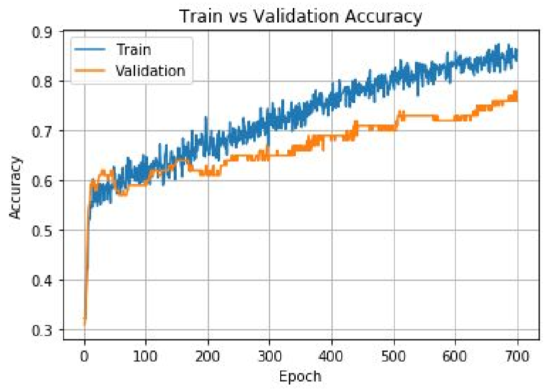
Final test accuracy: 0.9586206896551724

2) Batch size = 700

```
[ ] SGD(700, 700, 0, loss_type="MSE", learning_rate = 0.001, beta1=0.9, beta2=0.999, epsilon=1e-08)
```

Train vs Validation Loss/Accuracy Plots:





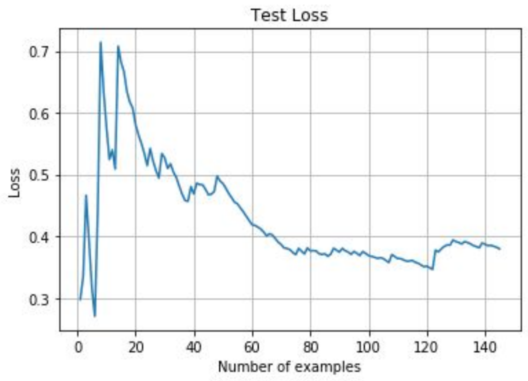
Final train loss: 0.20649094879627228

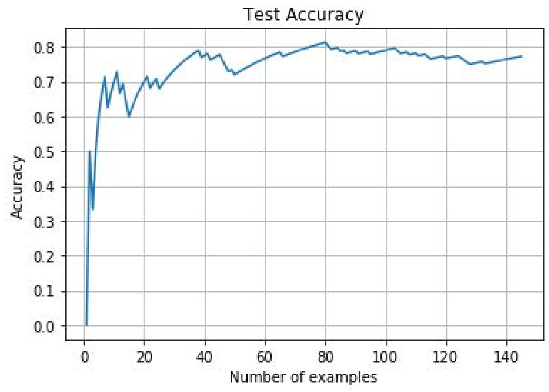
Final train accuracy: 0.8614285714285714

Final validation loss: 0.31817692518234253

Final validation accuracy: 0.76

Test Loss/Accuracy Plots:





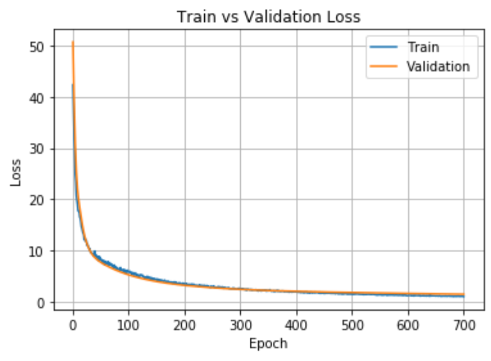
Final test loss: 0.37954544282208513

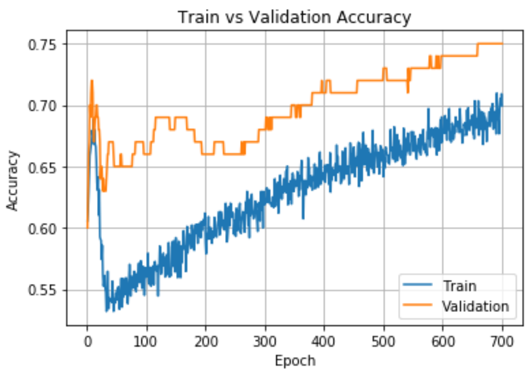
Final test accuracy: 0.7724137931034483

3) Batch size = 1750

```
[ ]  SGD(1750, 700, 0, loss_type="MSE", learning_rate = 0.001, beta1=0.9, beta2=0.999, epsilon=1e-08)
```

Train vs Validation Loss/Accuracy Plots:





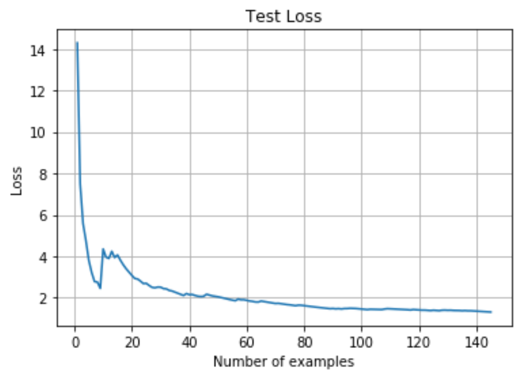
Final train loss: 0.9979153275489807

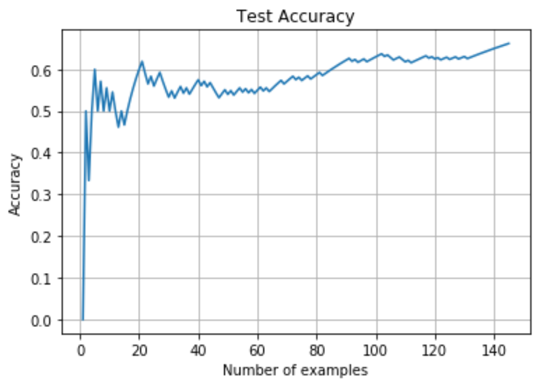
Final train accuracy: 0.6994285714285714

Final validation loss: 1.4632617235183716

Final validation accuracy: 0.75

Test Loss/Accuracy Plots:





Final test loss: 1.3110250143940687

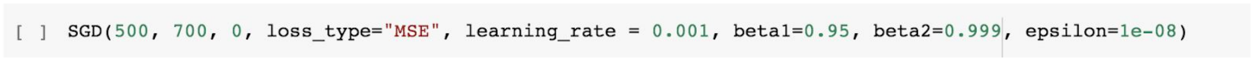
Final test accuracy: 0.6620689655172414

Batch Size	Train Accuracy	Validation Accuracy	Test Accuracy
100	1.0	0.97	0.9586206896551724
700	0.8614285714285714	0.76	0.7724137931034483
1750	0.6994285714285714	0.75	0.6620689655172414

A batch size of 100 has the highest accuracy for this model, 700 the second and 1750 the worst. The reason for this is that the lower batch size is, the more times of iteration the model can get trained in one epoch. In our problem, the more times of iteration give us a better result, but in other situations, the model may get overfitting.

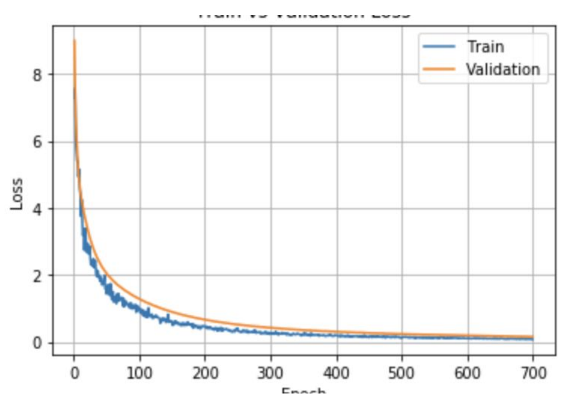
4. Hyperparameter Investigation

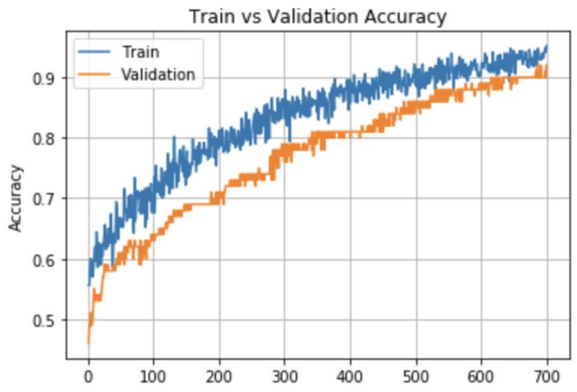
1) Beta1 = 0.95



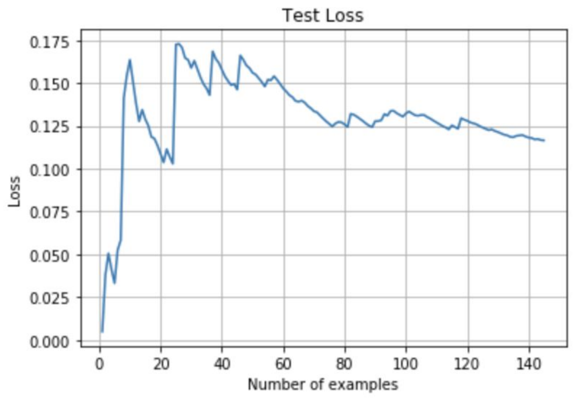
```
[ ]  SGD(500, 700, 0, loss_type="MSE", learning_rate = 0.001, beta1=0.95, beta2=0.999, epsilon=1e-08)
```

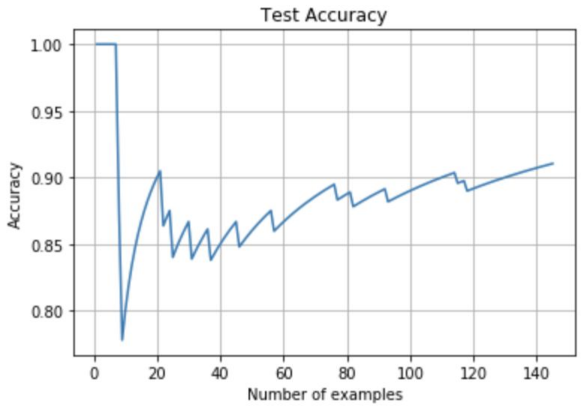
Train vs Validation Loss/Accuracy Plots:





Test Loss/Accuracy Plots:





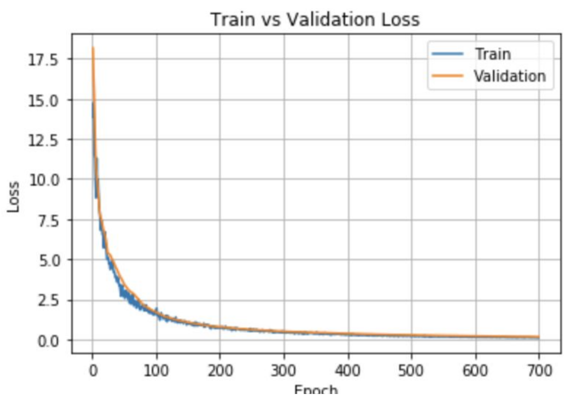
Final test loss: 0.1164906799076863

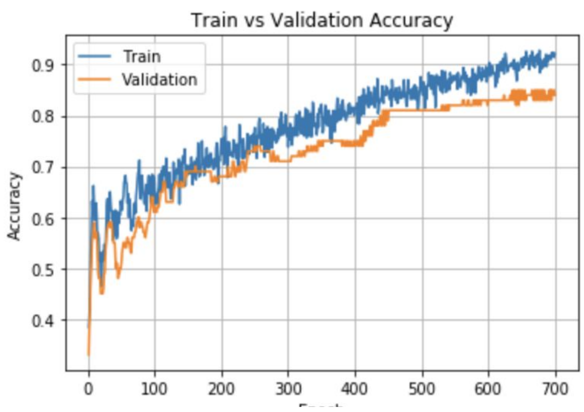
Final test accuracy: 0.9103448275862069

2) Beta1 = 0.99

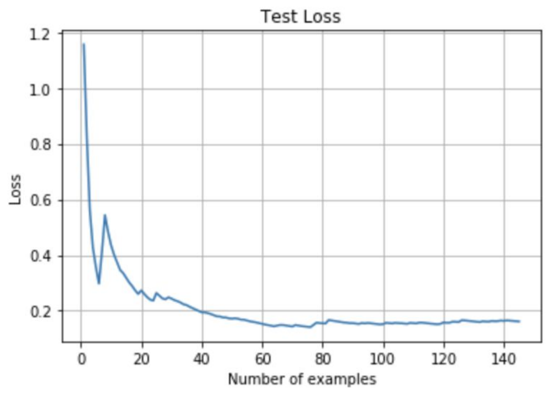
```
[ ] SGD(500, 700, 0, loss_type="MSE", learning_rate = 0.001, beta1=0.99, beta2=0.999, epsilon=1e-08)
```

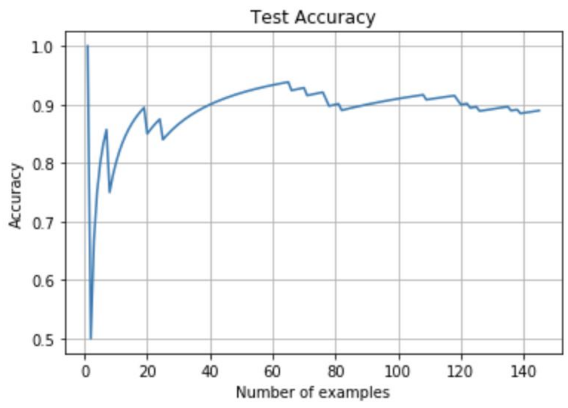
<u>Train vs Validation Loss/Accuracy Plots:</u>





<u>Test Loss/Accuracy Plots:</u>





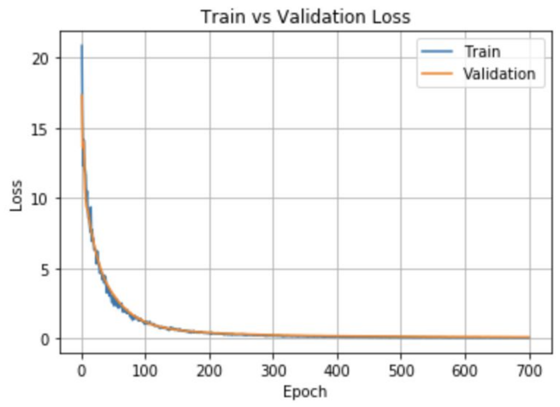
Final test loss: 0.16146351783390364

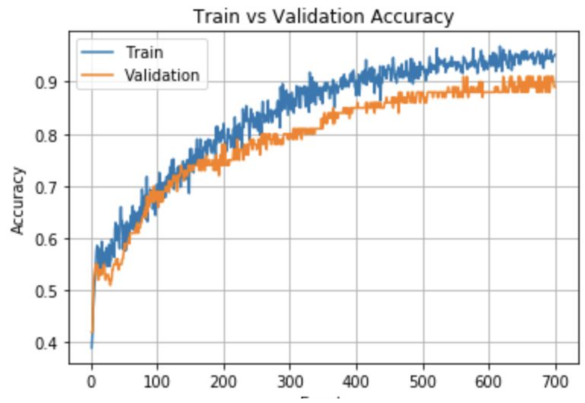
Final test accuracy: 0.8896551724137931

3) Beta2 = 0.99

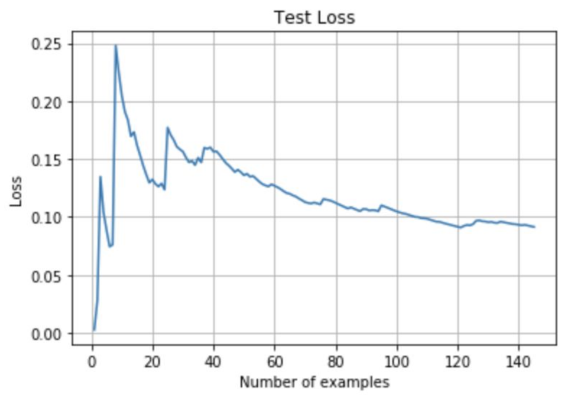
```
[ ] SGD(500, 700, 0, loss_type="MSE", learning_rate = 0.001, beta1=0.9, beta2=0.99, epsilon=1e-08)
```

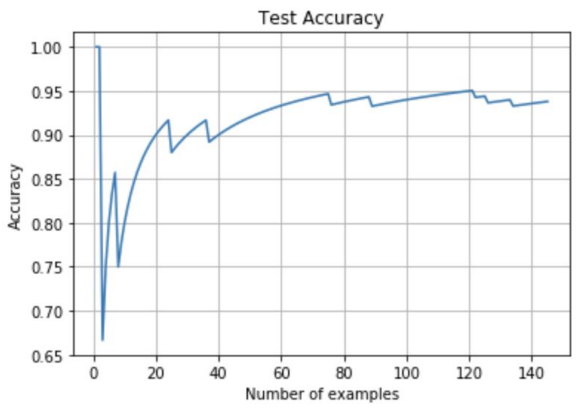
<u>Train vs Validation Loss/Accuracy Plots:</u>





Test Loss/Accuracy Plots:





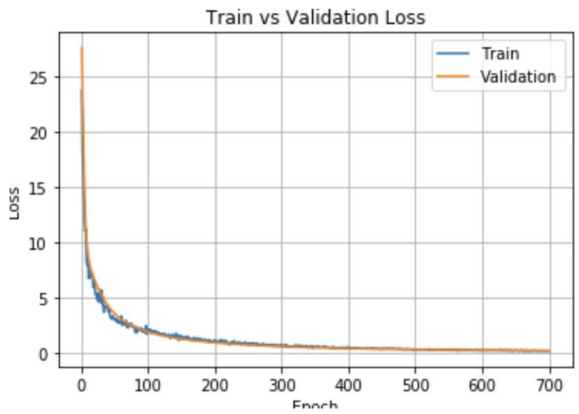
Final test loss: 0.0914032301720949

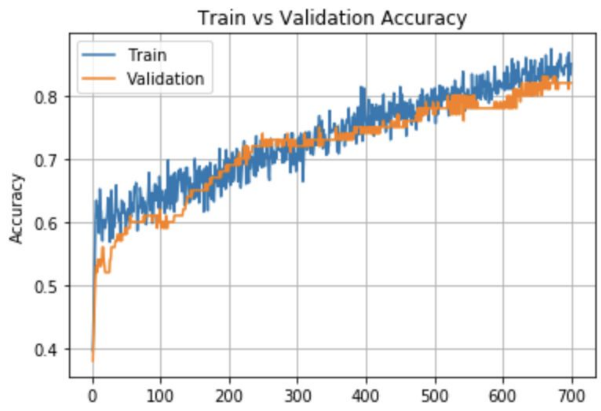
Final test accuracy: 0.9379310344827586

4) Beta2 = 0.9999

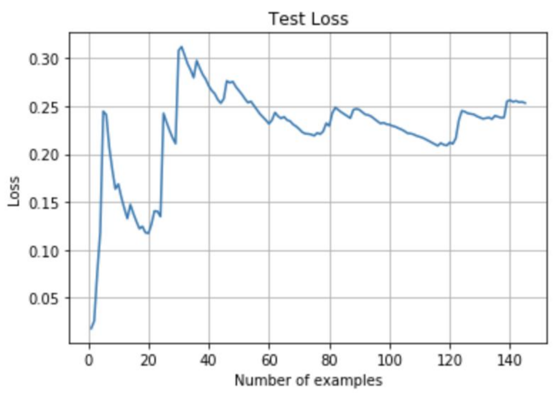
```
[ ]  SGD(500, 700, 0, loss_type="MSE", learning_rate = 0.001, beta1=0.9, beta2=0.9999, epsilon=1e-08)
```

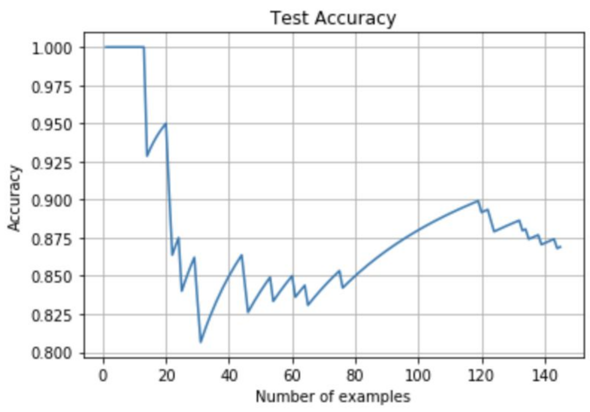
Train vs Validation Loss/Accuracy Plots:





Test Loss/Accuracy Plots:





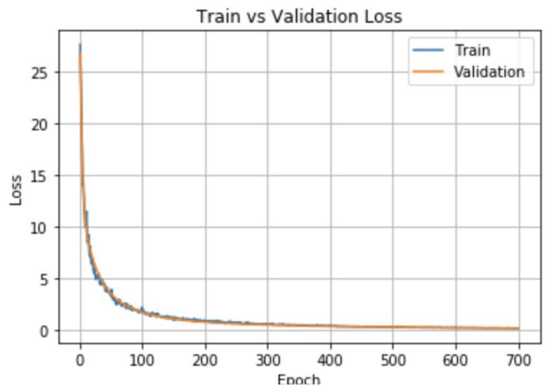
Final test loss: 0.25335430026831396

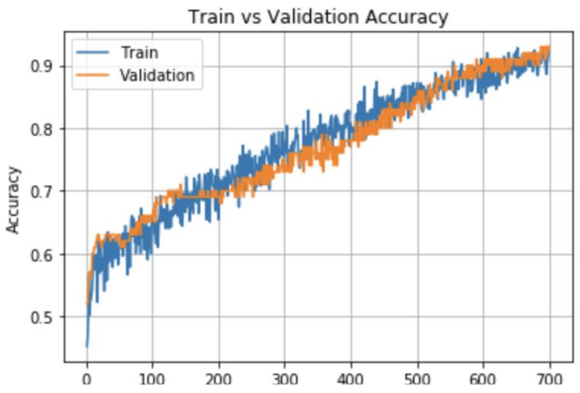
Final test accuracy: 0.8689655172413793

5) Epsilon = 1e-9

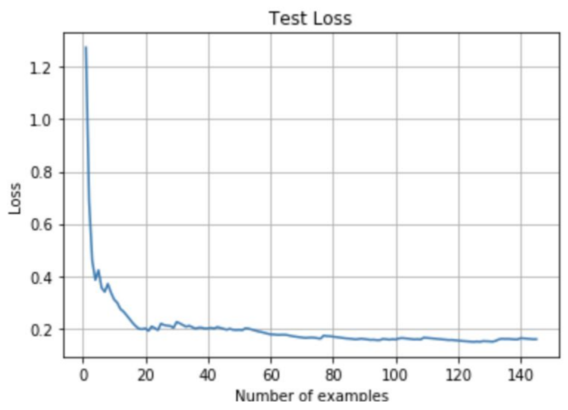
```
[ ] SGD(500, 700, 0, loss_type="MSE", learning_rate = 0.001, beta1=0.9, beta2=0.999, epsilon=1e-09)
```

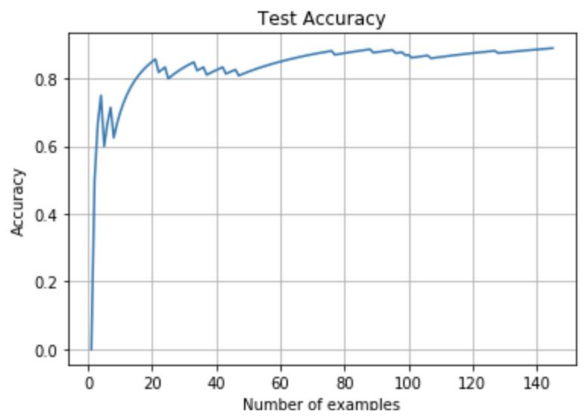
Train vs Validation Loss/Accuracy Plots:





Test Loss/Accuracy Plots:

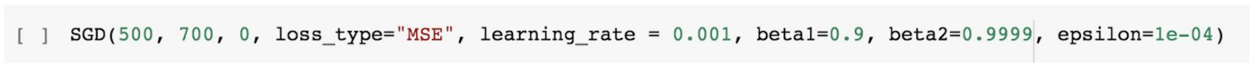




Final test loss: 0.1585115814275201

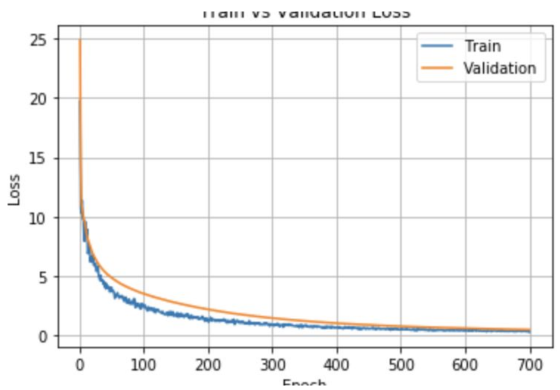
Final test accuracy: 0.8896551724137931

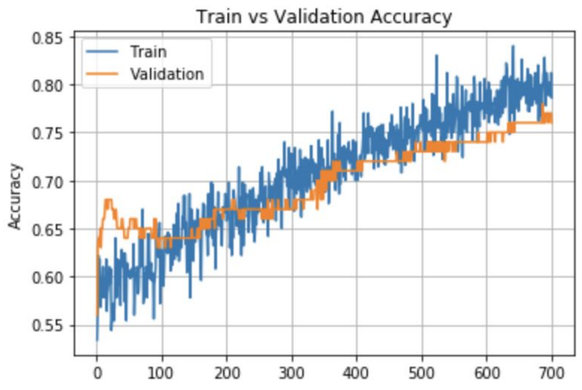
6) Epsilon = 1e-4



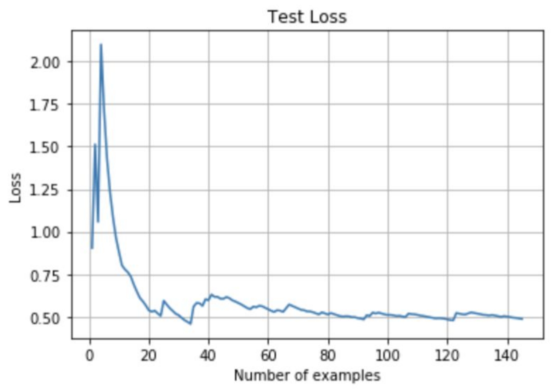
```
[ ] SGD(500, 700, 0, loss_type="MSE", learning_rate = 0.001, beta1=0.9, beta2=0.9999, epsilon=1e-04)
```

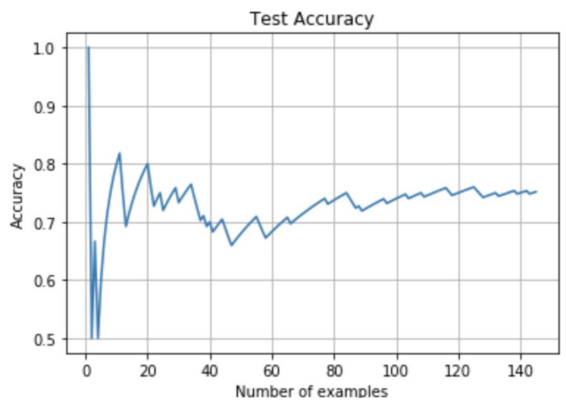
Train vs Validation Loss/Accuracy Plots:





Test Loss/Accuracy Plots:





Final test loss: 0.4864429380305975

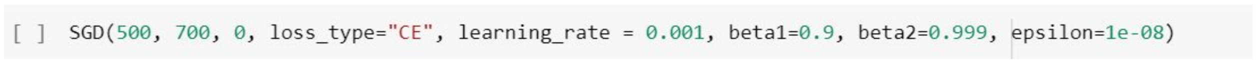
Final test accuracy: 0.7517241379310344

Beta 1 is the exponential decay rate for the first moment estimates. It measures how much momentum to add into consideration while doing gradient descent. Beta1 = 0.95 works better for our model. It has higher train, validation and test accuracy, and has lower train, validation and test MSE. But when Beta1 = 0.99, the momentum is too high and relies too much on previous values when computing the weight vector.
Beta 2 is the exponential decay rate for the second-moment estimates. It also measures how much momentum to add into consideration while doing gradient descent. Beta2 = 0.99 has higher train, validation and test accuracy. However, if Beta2 = 0.9999, the model will consider too much about historical values and thus affect the accuracy.
Epsilon is a very small number to prevent any division by zero in the implementation. When epsilon = 1e-9, our model gets a higher train, validation and test accuracy. It indicates 1e-9 is a proper value to consider zero division, while 1e-4 is too large that it affects the accuracy of our model.

5. Cross Entropy Loss Investigation

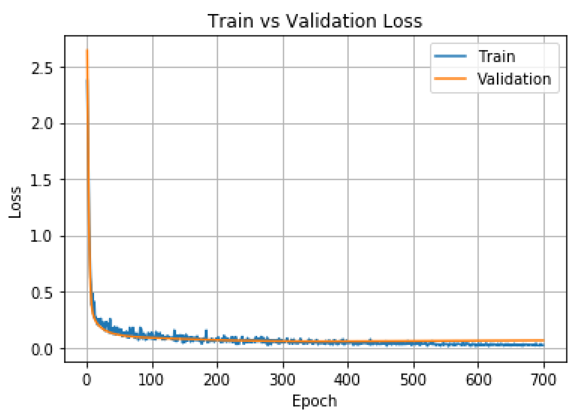
5.1. Implementing Stochastic Gradient Descent

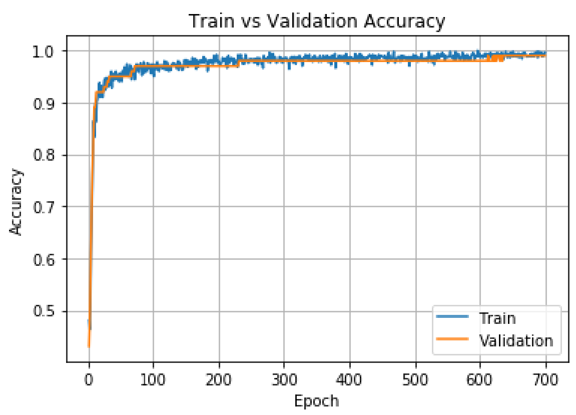
Run SGD with loss_type = “CE”, batch_size = 500, epochs = 700 and other parameters = default



```
[ ]  SGD(500, 700, 0, loss_type="CE", learning_rate = 0.001, beta1=0.9, beta2=0.999, epsilon=1e-08)
```

<u>Train vs Validation Loss/Accuracy Plots:</u>





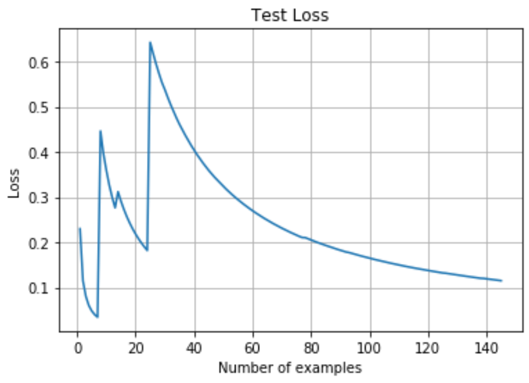
Final train loss: 0.020353475585579872

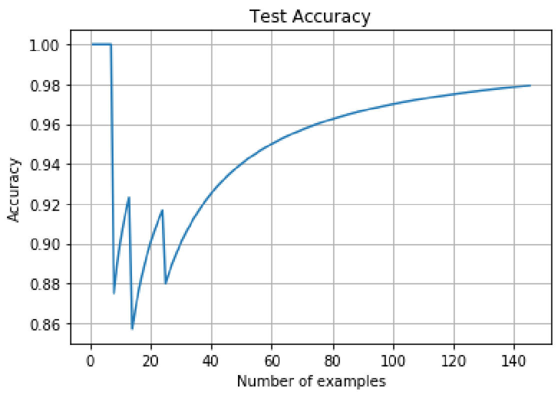
Final train accuracy: 0.996

Final validation loss: 0.0625944584608078

Final validation accuracy: 0.99

<u>Test Loss/Accuracy Plots:</u>



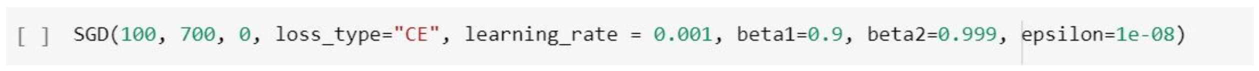


Final test loss: 0.11524920217922095

Final test accuracy: 0.979310344827586

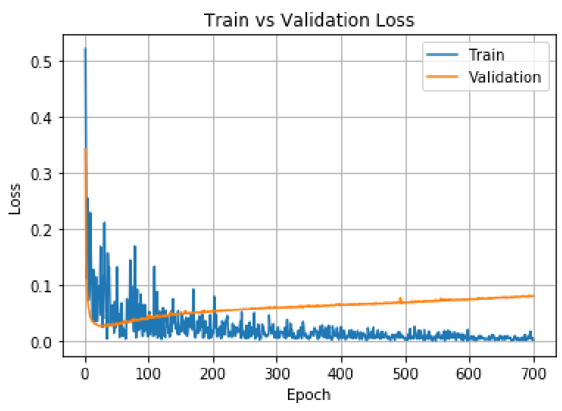
5.2. Batch Size Investigation

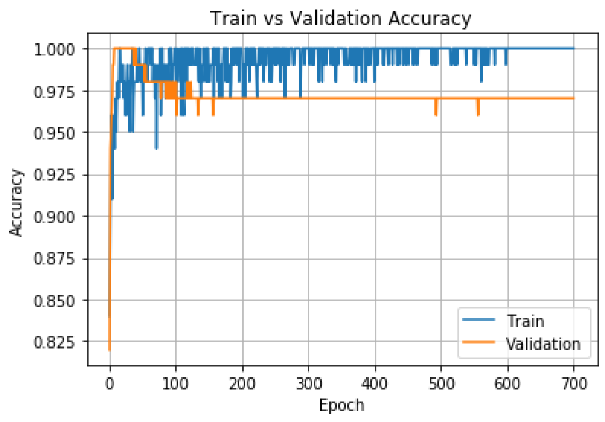
1) Batch size = 100



```
[ ]  SGD(100, 700, 0, loss_type="CE", learning_rate = 0.001, beta1=0.9, beta2=0.999, epsilon=1e-08)
```

<u>Train vs Validation Loss/Accuracy Plots:</u>





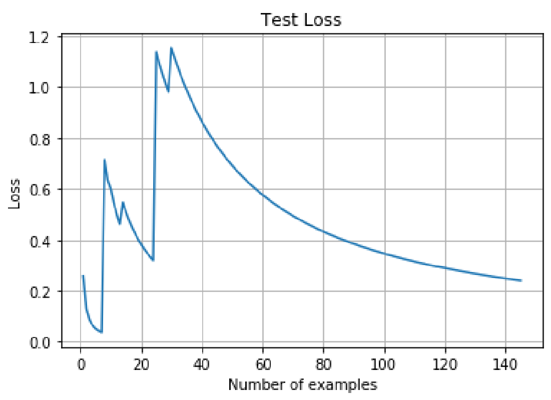
Final train loss: 0.0017431090818718076

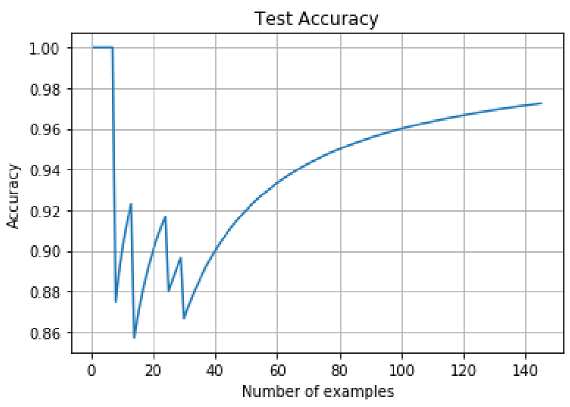
Final train accuracy: 1.0

Final validation loss: 0.08048097789287567

Final validation accuracy: 0.97

Test Loss/Accuracy Plots:





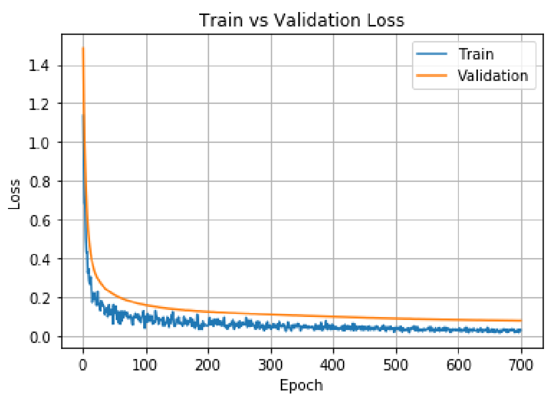
Final test loss: 0.2401311112372014

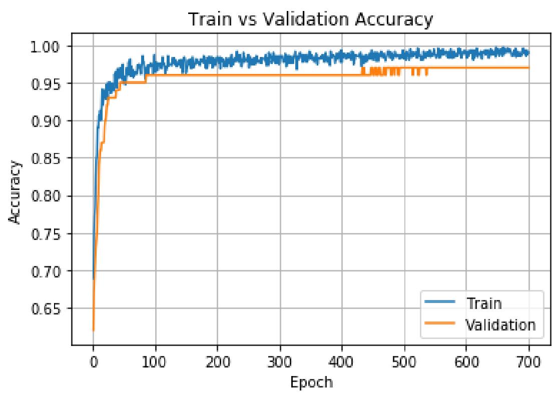
Final test accuracy: 0.9724137931034482

2) Batch size = 700

```
[ ]  SGD(700, 700, 0, loss_type="CE", learning_rate = 0.001, beta1=0.9, beta2=0.999, epsilon=1e-08)
```

Train vs Validation Loss/Accuracy Plots:





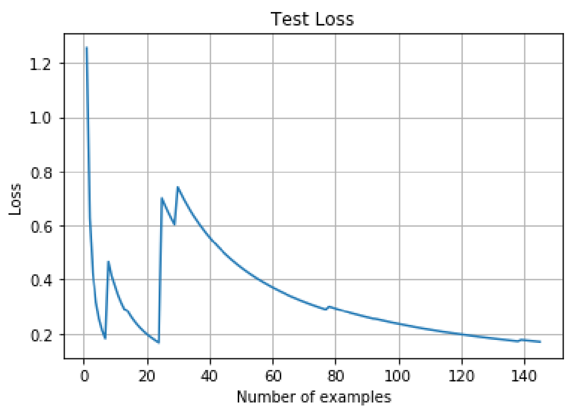
Final train loss: 0.029430737718939 78

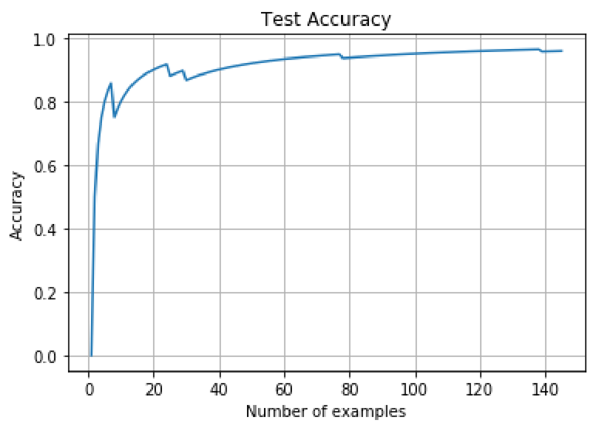
Final train accuracy: 0.99

Final validation loss: 0.0788288488984108

Final validation accuracy: 0.97

<u>Test Loss/Accuracy Plots:</u>





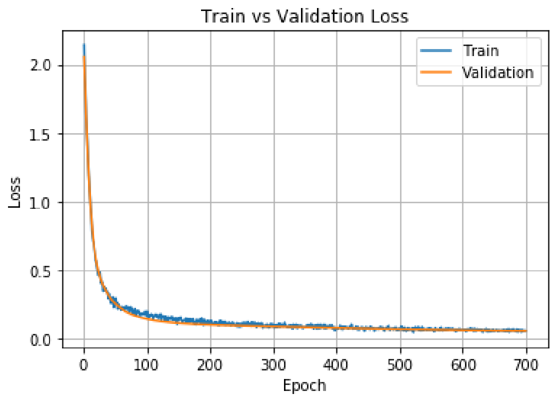
Final test loss: 0.1693440682742414

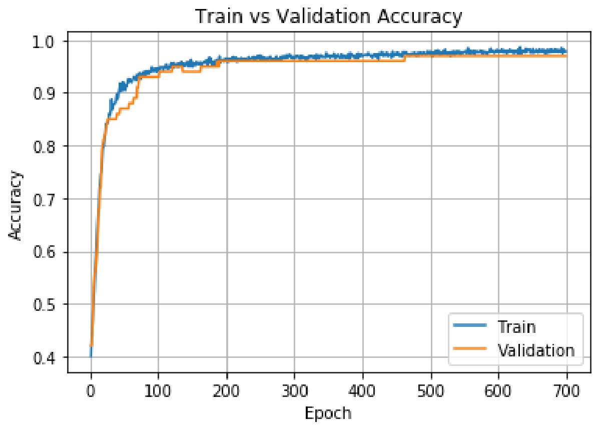
Final test accuracy: 0.9586206896551724

3) Batch size = 1750

```
[ ]  SGD(1750, 700, 0, loss_type="CE", learning_rate = 0.001, beta1=0.9, beta2=0.999, epsilon=1e-08)
```

<u>Train vs Validation Loss/Accuracy Plots:</u>





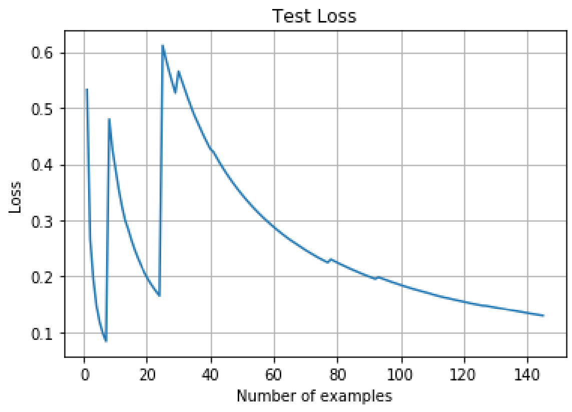
Final train loss: 0.05415783450007439

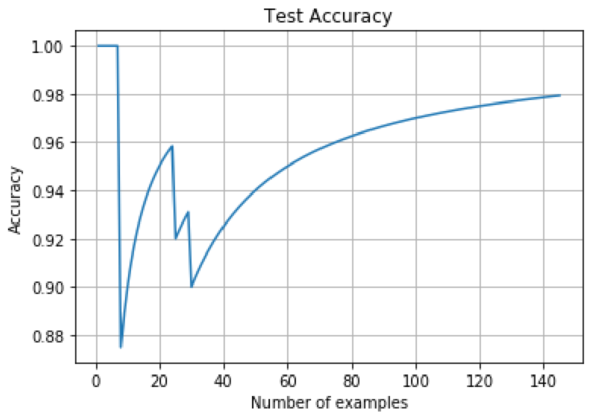
Final train accuracy: 0.9782857142857143

Final validation loss: 0.054990366101264954

Final validation accuracy: 0.97

Test Loss/Accuracy Plots:





Final test loss: 0.12983850655959095

Final test accuracy: 0.9793103448275862

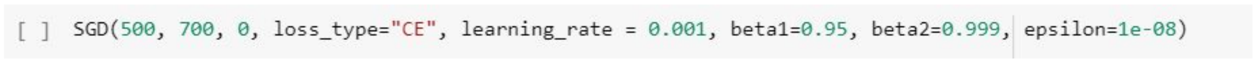
Batch Size	**Train Accuracy**	**Validation Accuracy**	**Test Accuracy**
100	1.0	0.97	0.9724137931034482

700	0.99	0.97	0.9586206896551724
1750	0.9782857142857143	0.97	0.9793103448275862

For cross entropy loss, all the three batch sizes achieve similar accuracy. Larger batch sizes provide a less noisy model, but take more epochs to converge since they require more computation to complete one update.

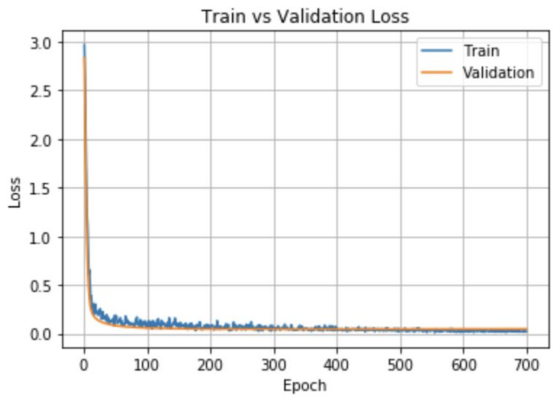
5.3. Hyperparameter investigation

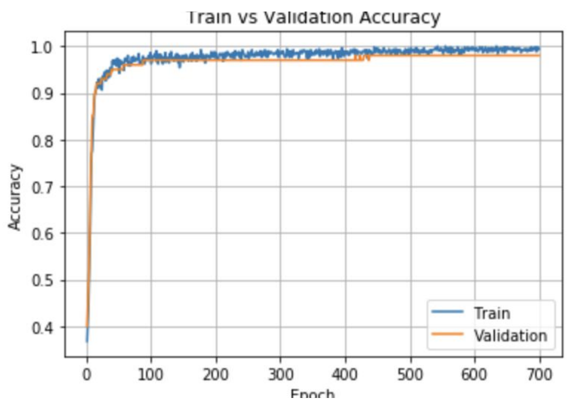
1) Beta1 = 0.95



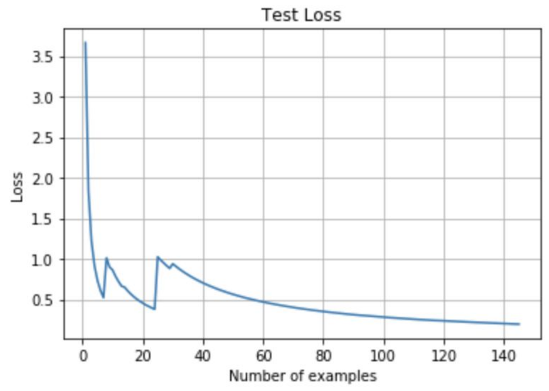
```
[ ]  SGD(500, 700, 0, loss_type="CE", learning_rate = 0.001, beta1=0.95, beta2=0.999, epsilon=1e-08)
```

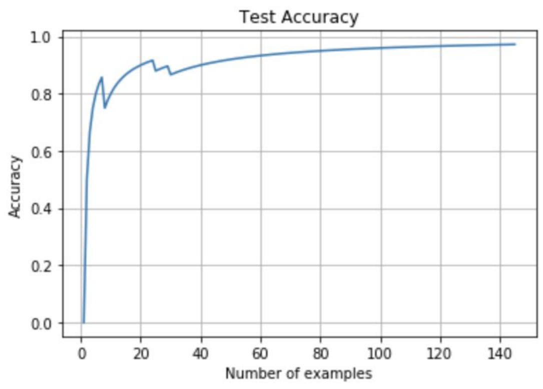
Train vs Validation Loss/Accuracy Plots:





Test Loss/Accuracy Plots:

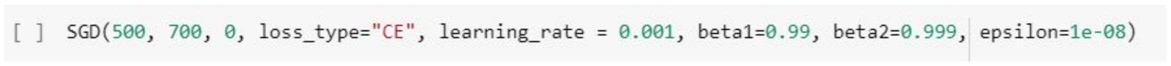




Final test loss: 0.1304270858635817

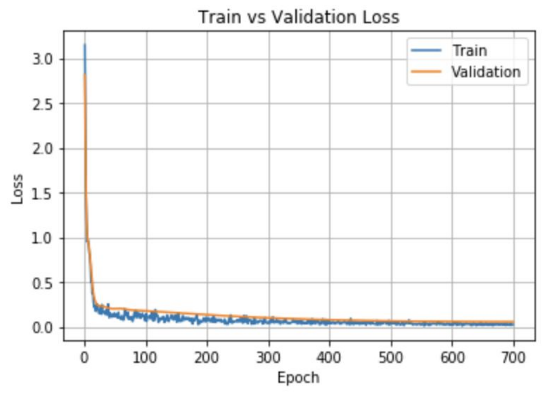
Final test accuracy: 0.9724137931034482

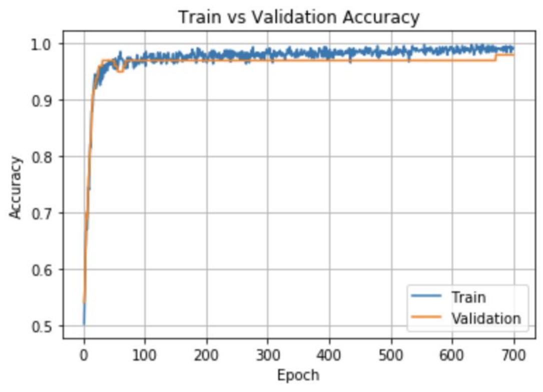
2) Beta1 = 0.99



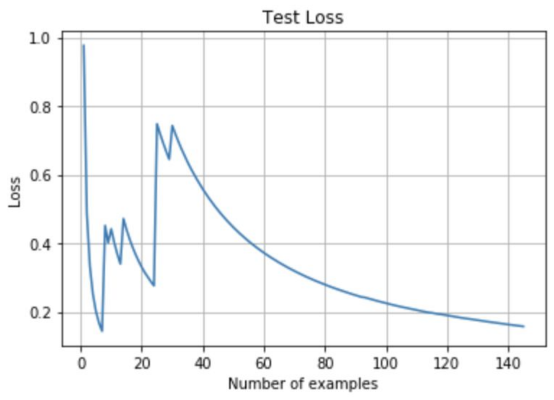
```
[ ] SGD(500, 700, 0, loss_type="CE", learning_rate = 0.001, beta1=0.99, beta2=0.999, epsilon=1e-08)
```

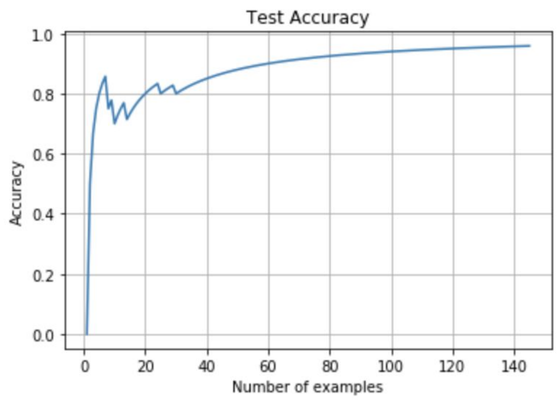
<u>Train vs Validation Loss/Accuracy Plots:</u>





<u>Test Loss/Accuracy Plots:</u>

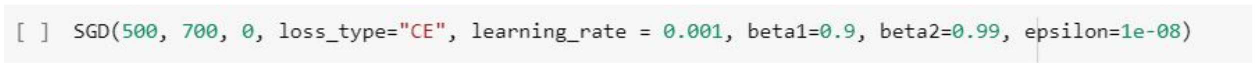




Final test loss: 0.15748197296016078

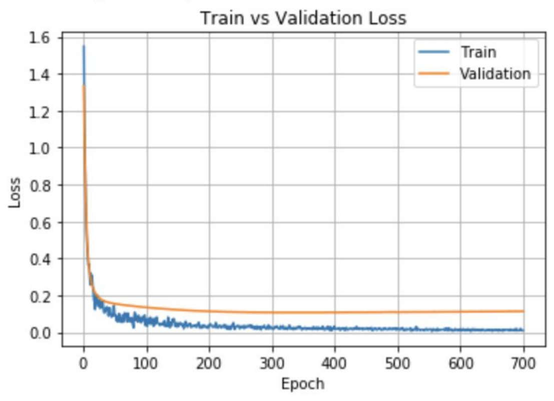
Final test accuracy: 0.9586206896551724

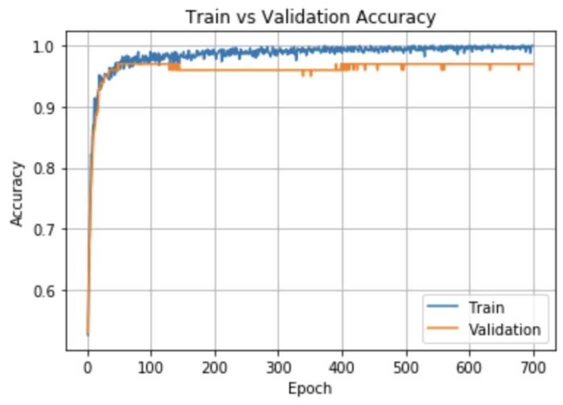
3) Beta2 = 0.99



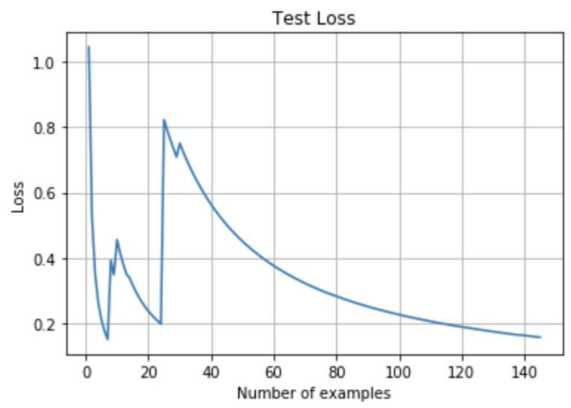
```
[ ] SGD(500, 700, 0, loss_type="CE", learning_rate = 0.001, beta1=0.9, beta2=0.99, epsilon=1e-08)
```

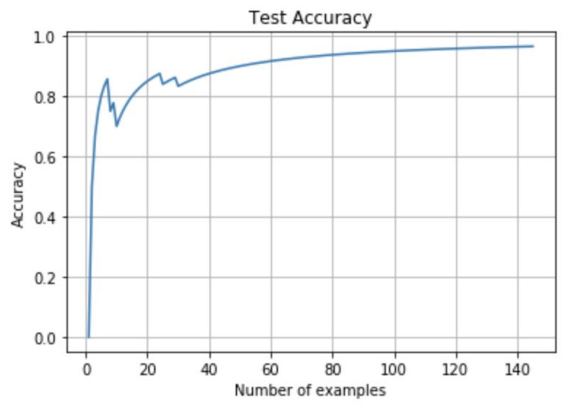
<u>Train vs Validation Loss/Accuracy Plots:</u>





<u>Test Loss/Accuracy Plots:</u>

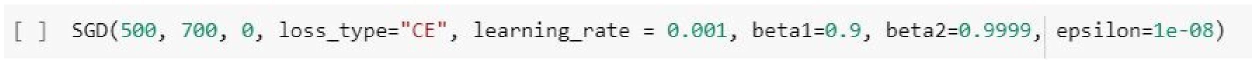




Final test loss: 0.15680019945530224

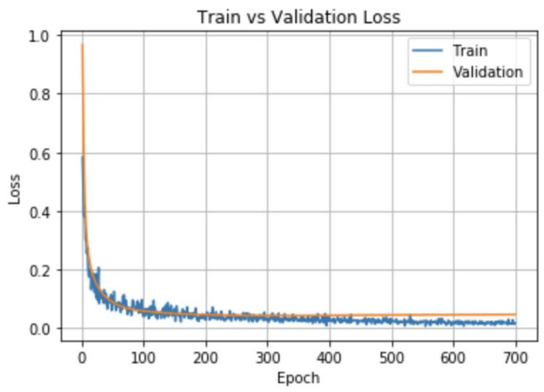
Final test accuracy: 0.9655172413793104

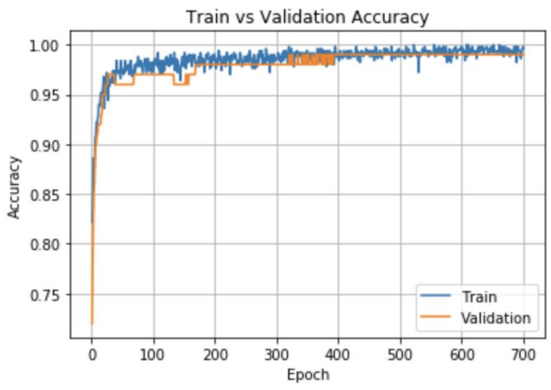
4) Beta2 = 0.9999



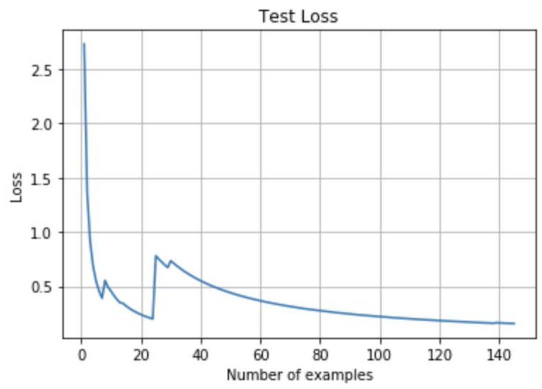
```
[ ]  SGD(500, 700, 0, loss_type="CE", learning_rate = 0.001, beta1=0.9, beta2=0.9999, epsilon=1e-08)
```

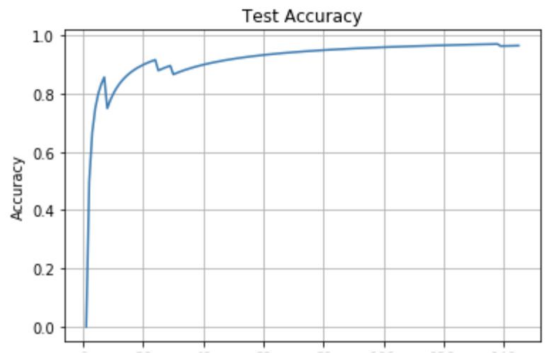
<u>Train vs Validation Loss/Accuracy Plots:</u>





Test Loss/Accuracy Plots:

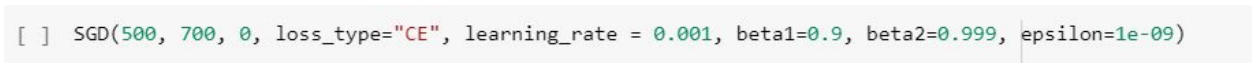




Final test loss: 0.1585471037038797

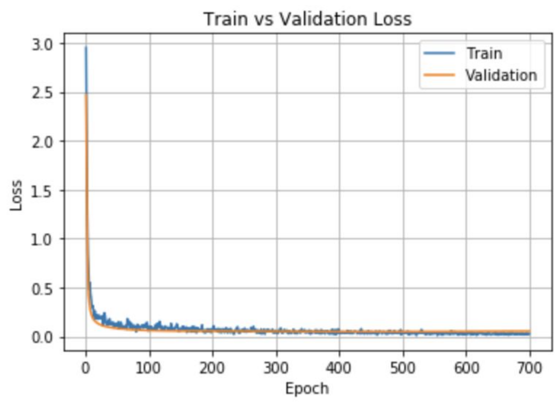
Final test accuracy: 0.9655172413793104

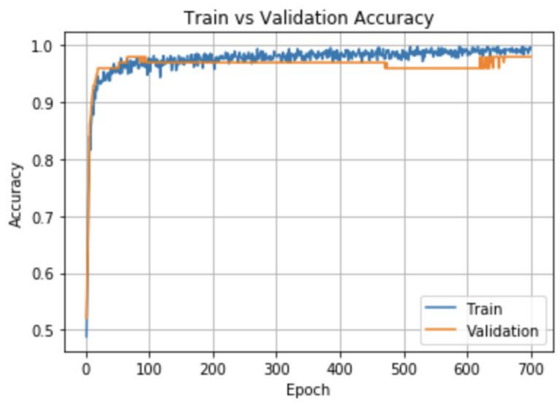
5) Epsilon = 1e-9



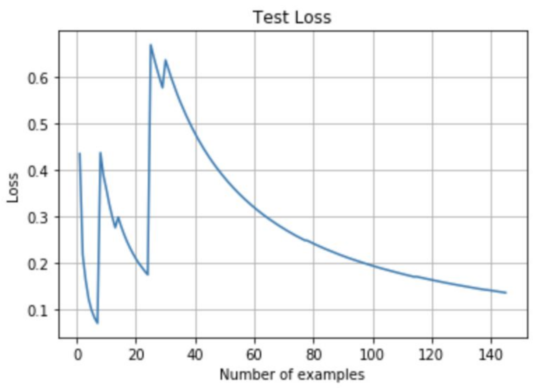
```
[ ]  SGD(500, 700, 0, loss_type="CE", learning_rate = 0.001, beta1=0.9, beta2=0.999, epsilon=1e-09)
```

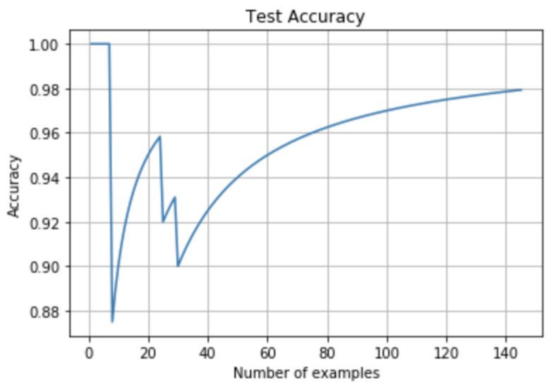
Train vs Validation Loss/Accuracy Plots:





<u>Test Loss/Accuracy Plots:</u>

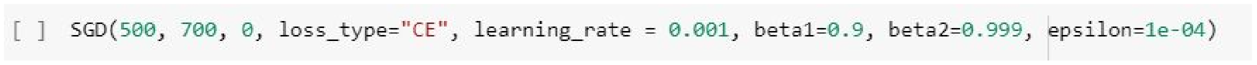




Final test loss: 0.13575129436528266

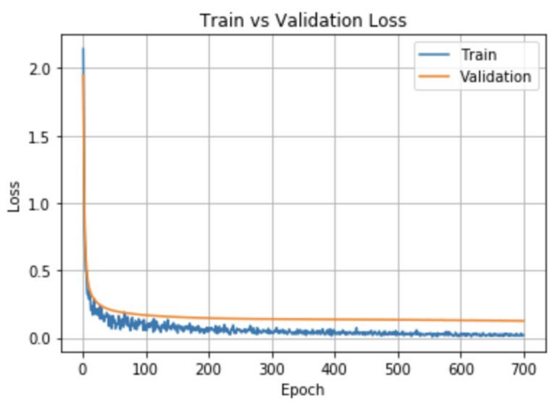
Final test accuracy: 0.9793103448275862

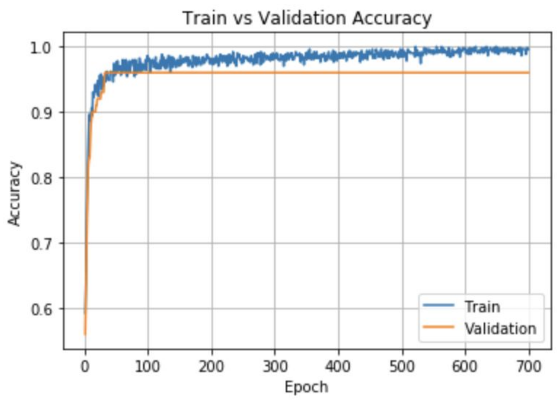
6) Epsilon = 1e-4



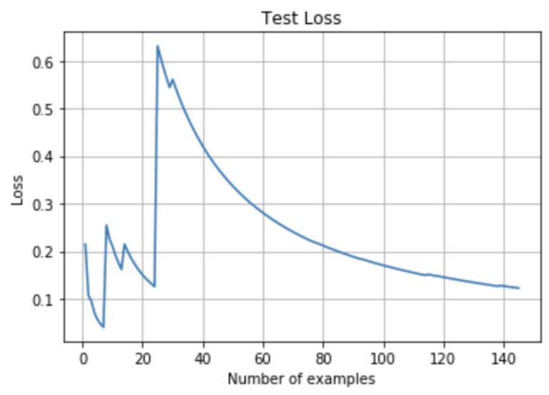
```
[ ]  SGD(500, 700, 0, loss_type="CE", learning_rate = 0.001, beta1=0.9, beta2=0.999, epsilon=1e-04)
```

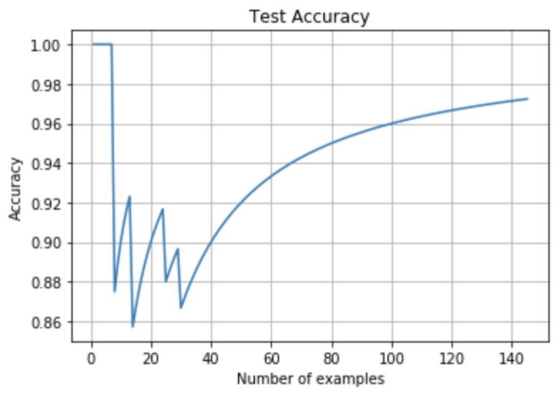
<u>Train vs Validation Loss/Accuracy Plots:</u>





Test Loss/Accuracy Plots:





Final test loss: 0.12246414795344814

Final test accuracy: 0.9724137931034482

For cross entropy loss, different hyperparameters produce similar accuracy, indicating a more stable model. Based on the plots, CE can achieve convergence in less number of epochs and the can produce a less noisy model.

Comparison with SGD with MSE:

Methods	Test accuracy (best)	Test loss (best)
SGD with MSE	0.9379310344827586	0.0914032301720949
SGD with CE	0.9793103448275862	0.13575129436528266

As shown in the table, the CE is a better loss function for SGD in our problem. It achieved higher overall accuracy. And the plots of SGD with CE are less noisy than the plots of SGD with MSE and achieved faster convergence.

6. Comparison against Batch GD

Methods	**Test accuracy (best)**	**Test loss (best)**
Batch GD with MSE	0.9724137931034482	0.03540431112847913
SGD with MSE	0.9379310344827586	0.0914032301720949
Batch GD with CE	0.9793103448275862	0.08846602141763325
SGD with CE	0.9793103448275862	0.13575129436528266

We chose the test accuracy and test loss as criterias to compare between two models because the test dataset gives us unbiased evaluation about the performance of models.

Batch GD with MSE reaches higher test accuracy and lower test loss than SGD with MSE, thus the Batch GD with MSE is a better model. The reason might be that SGD fails to go through all data because of the reshuffle.

However, Batch GD with CE and SGD with CE have around the same performance. The test accuracy are both very high, which indicates CE is the better loss function for our problem.
In terms of the plots, the plots of SGD are obviously more noisy than those of Batch GD, especially in the cases of MSE. It is probably caused by the reshuffle of train data, which introduces unstability to the model.