Name: Venkata Meghana Achanta

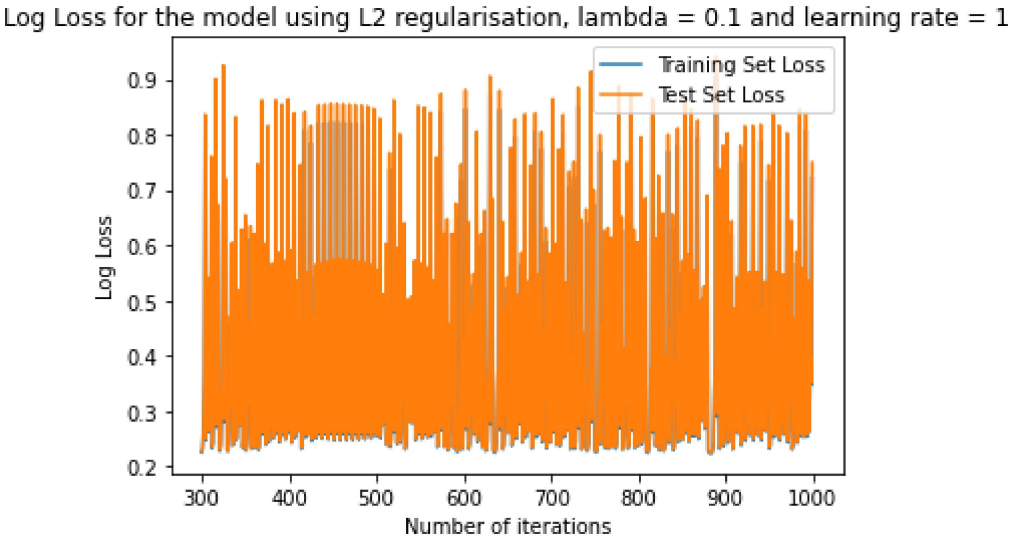
USC ID: 2578990261

Q1a)

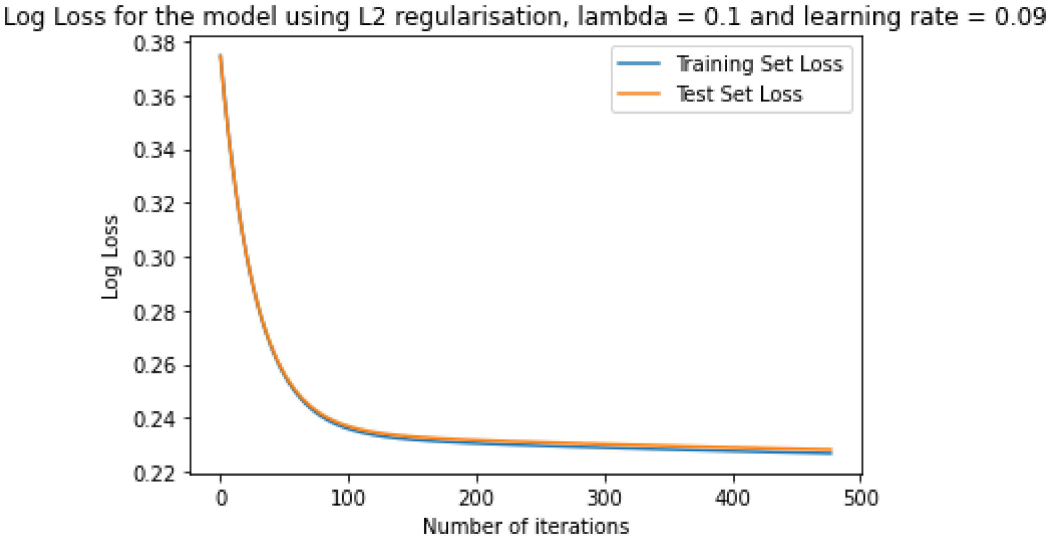
i.

To determine my learning rate, I experimented with three different learning rates: 1,0.09 and 0.00001.

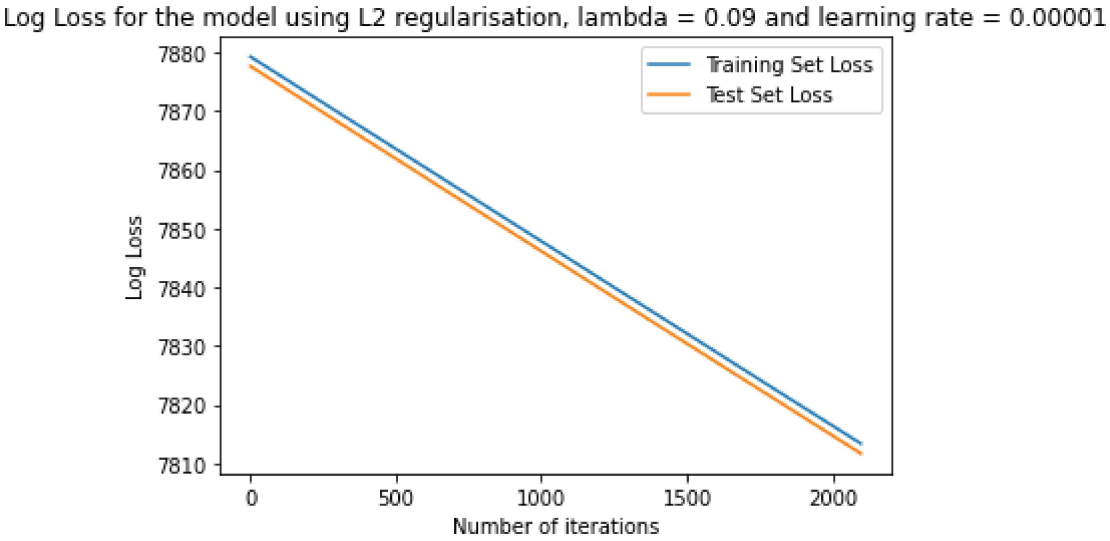
When I tried to train the model with 1 as learning rate, the loss oscillated between a range of values but never achieved convergence.



Next, I trained the model with learning rate 0.09. The number of iterations required were adequate and the convergence was obtained in required time.



While training the model with learning rate 0.00001, the model did not converge for almost 2000 iterations but if we wait for a much longer time, it might achieve convergence. But this is not optimal in real world scenarios.



ii.

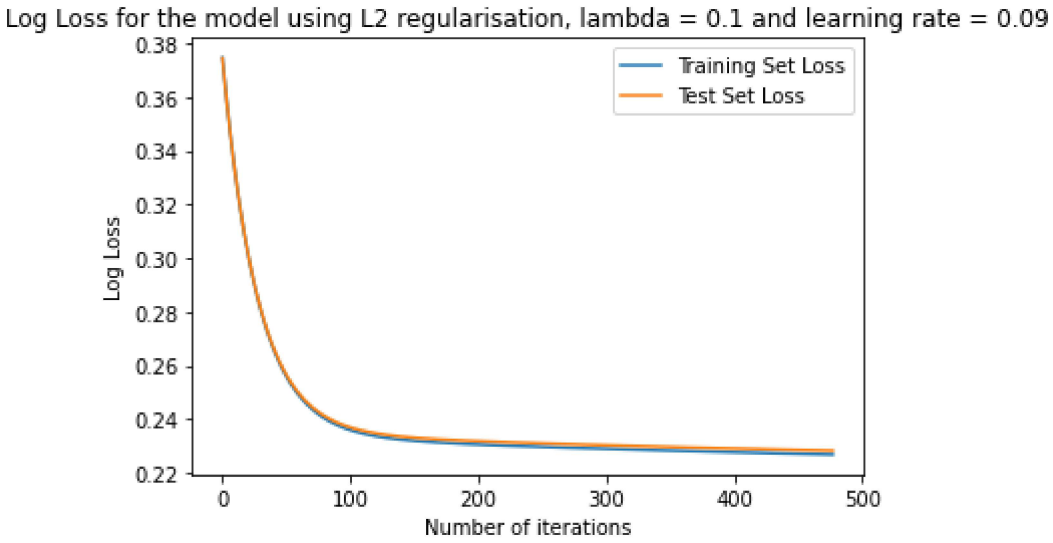
To obtain model convergence, I used the condition that the model should be continued to train till the difference between the absolute loss values is greater than the order of 10^(-5). Once the difference between consecutive loss values went less than the order of (10^-5), it should stop iterating and the weights will be saved.

iii.

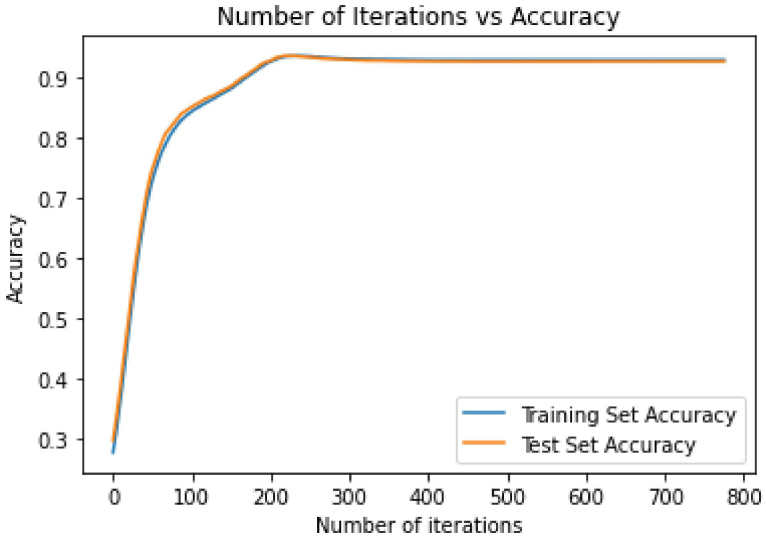
I used L2 Regularization in my model. I tried various values of lambda such as 1, 0.1 and 0.00001. After experimenting, I understood that the lambda value of 0.1 ensured that the weights did not explode and ensured that the model achieved convergence in fewer iterations.

iv.

The plot of Learning Curve for log loss of training and test set is as follows:



Accuracy of the training and test set is as follows:

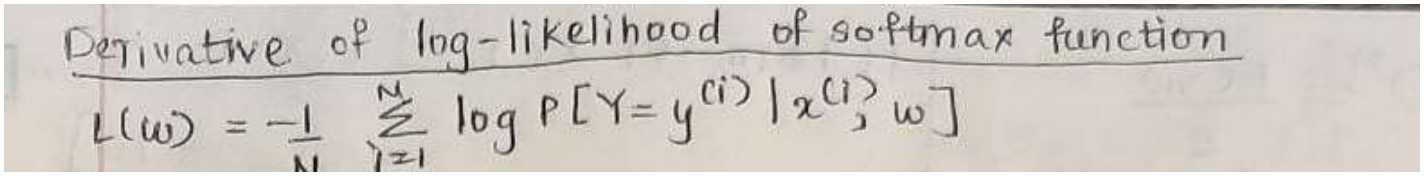


v. I trained my model twice, the first time, I initialised my weights and bias as zeroes which gave me a training accuracy of 97.63% and testing accuracy of 97%.

The second time, I initialised my weights and biases by using the normal distribution with mean 0 and standard deviation 10. The final training accuracy for my model is about 93% and the testing accuracy is about 92.78%. The final loss obtained for the training set is 0.226797 and the final loss obtained for the testing set is 0.2282166.

Q1.b

i.



$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial w}$$

where $a = P[Y=1 \mid x, w] = \frac{e^{w_1^T x}}{\sum_{j=0}^{K} e^{w_j^T x}}$ $\quad a = w_1^T x$

$$\frac{\partial a}{\partial w} = \frac{\partial}{\partial w}[w_1^T x] = w_1^T x$$

$$\frac{\partial L}{\partial a} = -\frac{1}{N} \sum_{i=1}^{N} \frac{\partial}{\partial a_i}\left[\log\left[\frac{e^{a_j}}{\sum_{j=1}^{K} e^{a_j}}\right]\right]$$

$$= -\frac{1}{N} \sum_{i=1}^{N} \frac{\partial}{\partial a_i}\left[\log e^{a_j} - \log \sum_{j=1}^{K} e^{a_j}\right]$$

$$= -\frac{1}{N} \sum_{i=1}^{N} \left[\frac{\partial(a_j)}{\partial a_i} - \frac{\partial}{\partial a_i}\left[\log \sum_{j=1}^{K} e^{a_j}\right]\right]$$

if $i=j$

$$= -\frac{1}{N} \sum_{i=1}^{N} \left[1 - \frac{1}{\sum_{j=1}^{K} e^{a_j}} \frac{\partial}{\partial a_i}\left[\sum_{j=1}^{K} e^{a_j}\right]\right]$$

$$= -\frac{1}{N} \sum_{i=1}^{N} \left[1 - \frac{1}{\sum_{j=1}^{K} e^{a_j}} \left[\sum_{j=1}^{K} \frac{\partial e^{a_j}}{\partial a_i}\right]\right]$$

$$= -\frac{1}{N} \sum_{i=1}^{N} \left[1 - \frac{1}{\sum_{j=1}^{K} e^{a_j}} \left[\sum_{j=1}^{K} e^{a_j} \frac{\partial a_j}{\partial a_i}\right]\right]$$

if $i=j$;

$$\frac{\partial L}{\partial a} = -\frac{1}{N} \sum_{i=1}^{N} \left[1 - P[Y=y^{(i)} \mid x^{(i)}, w]\right]$$

$$\frac{\partial L}{\partial} = \left[-\frac{1}{N} \sum_{i=1}^{N} \left[1 - P[Y=y^{(i)} \mid x^{(i)}, w]\right]\right] \cdot w x$$

$$\frac{\partial}{\partial w}\left[\sum_{i=1}^{N} \quad \right]$$

for $i \neq j$

$$\frac{\partial L}{\partial a} = \frac{\partial}{\partial a}\left[-\frac{1}{N}\sum_{i=1}^{N} \log\left[\frac{e^{a_j}}{\sum_{j=1}^{K} e^{a_j}}\right]\right]$$

$$= -\frac{1}{N}\sum_{i=1}^{N} \frac{\partial}{\partial a_i}\left[\log\left[\frac{e^{a_j}}{\sum_{j=1}^{K} e^{a_j}}\right]\right]$$

$$= -\frac{1}{N}\sum_{i=1}^{N}\left[\frac{\partial a_j}{\partial a_i} - \frac{\partial}{\partial a_i}\log\left[\sum_{j=1}^{K} e^{a_j}\right]\right]$$

$$= -\frac{1}{N}\sum_{i=1}^{N}\left[-\frac{\partial}{\partial a_i}\log\left[\sum_{j=1}^{K} e^{a_j}\right]\right]$$

$$= \frac{1}{N}\sum_{i=1}^{N}\left[\frac{+\, e^{a_j}}{\sum_{j=1}^{K} e^{a_j}}\right]$$

$$= -P[Y = y^{(i)} \mid x^{(i)}, w]$$

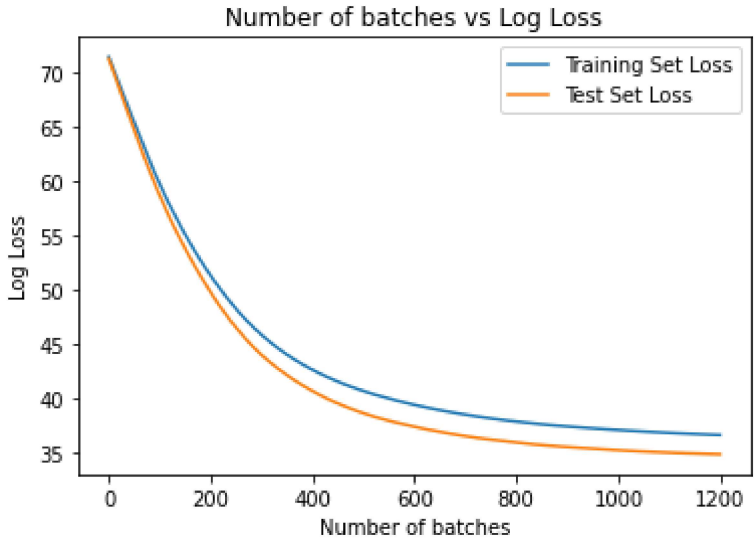
$$\frac{\partial L}{\partial w} = -P[Y = y^{(i)} \mid x^{(i)}, w] \cdot x$$

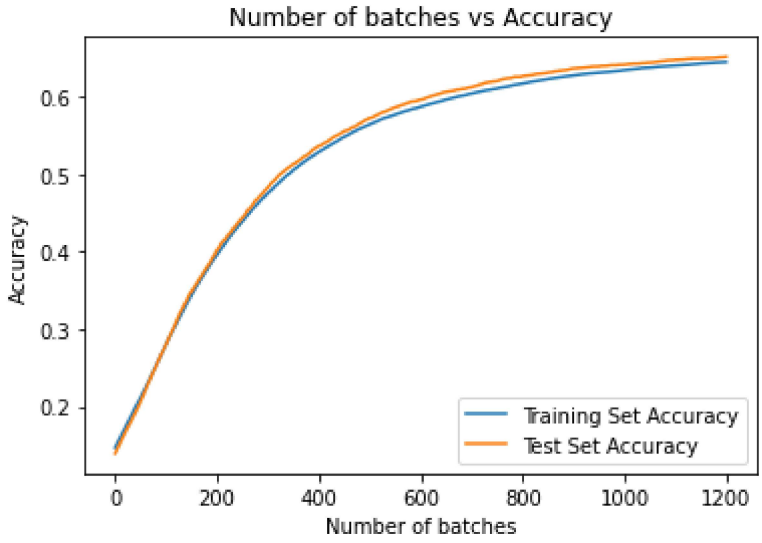
$$\therefore \frac{\partial L}{\partial w} = \begin{cases} x\left[1 - P[Y = y^{(i)} \mid x^{(i)}, w\right] & ; i = j \\ -P[Y = y^{(i)} \mid x^{(i)}, w] \cdot x & ; i \neq j \end{cases}$$

ii. To train the model, I used a learning rate of 0.1 as it gave optimum convergence. Using a lower learning rate such as 0.00001 will make the convergence slower which is not feasible. For higher learning rates, the model will not obtain convergence and it would oscillate.

For my model, I used a random seed value to initialize my weights. For different seed values, the loss obtained was very different.

iii. The plots for accuracy and learning curves is as follows:

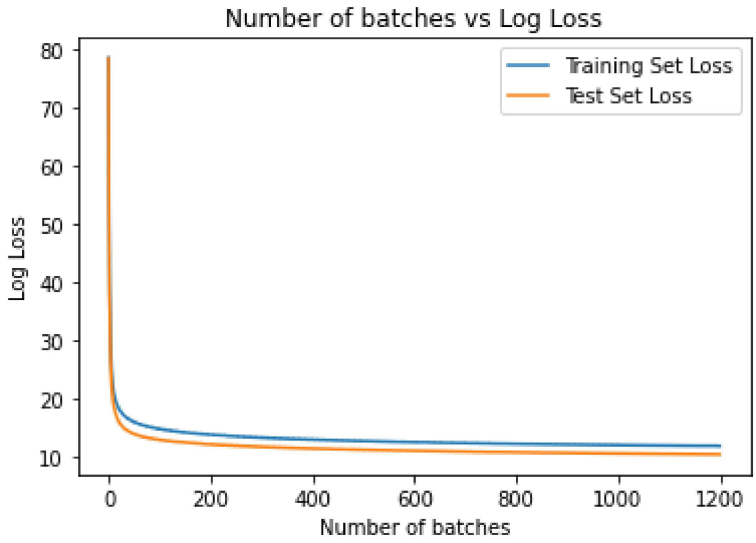


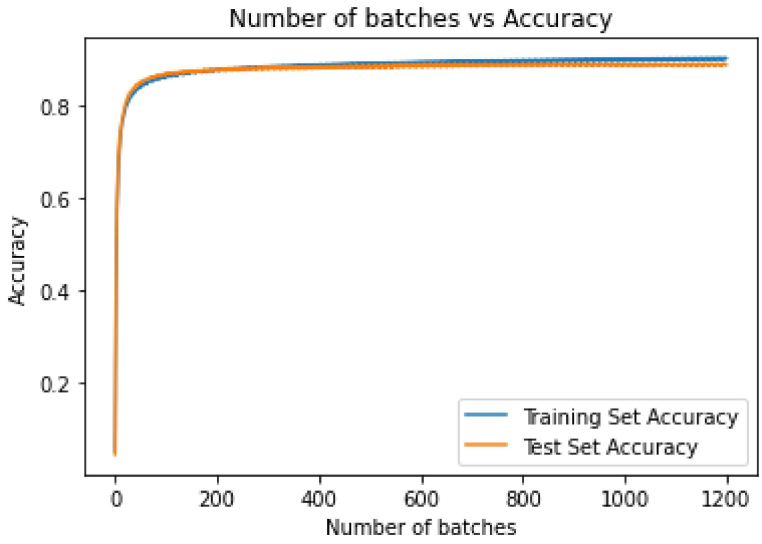


iv. For my model, the training loss obtained is 38.700472, the testing loss obtained is 36.833, the accuarcy for the training set is 61.33% and the accuracy for the testing set is 62.24%.

Q1c

i. The plot for mini-batch size 1 for loss and accuracy is as follows:



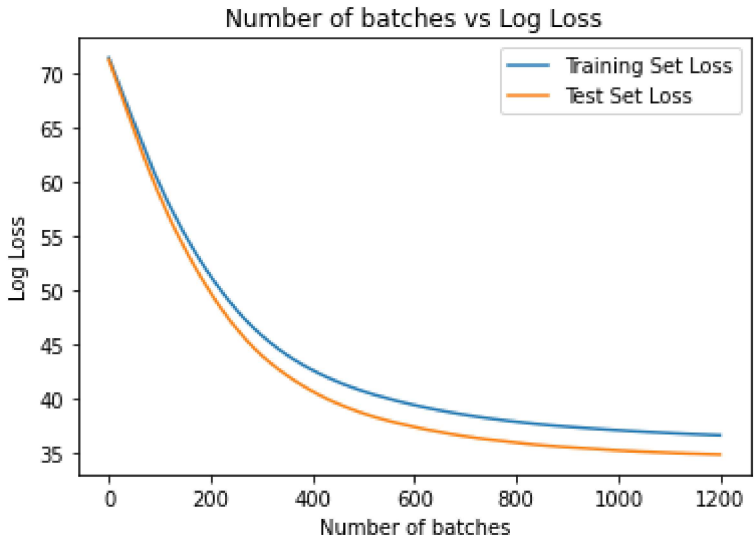


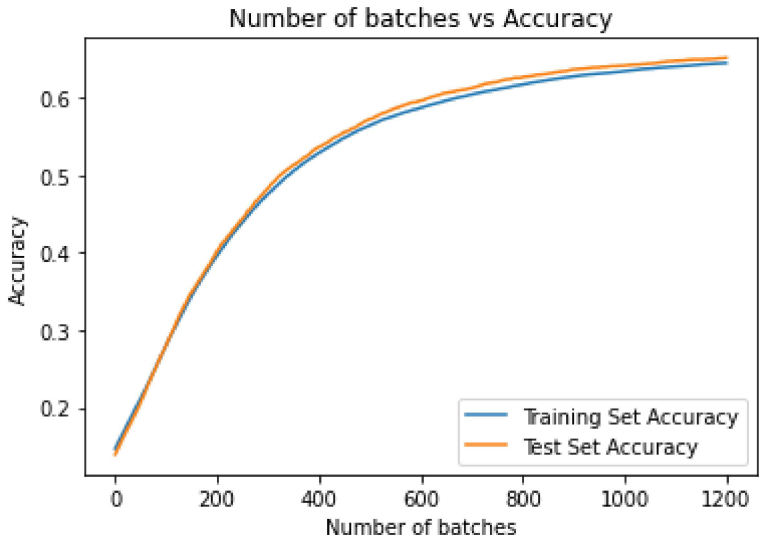
With a mini-batch size of 1, it took 100 epochs to get an accuracy of 89% whereas for batch gradient descent it took about 1000 epochs to get an accuracy of 62%. For the batch gradient descent to have similar outcome as mini batch 1, it might need to be run for few more epochs. The number of epochs depends on the learning rate. A higher learning rate ensures that batch gradient descent achieves convergence faster however, a lower learning rate is sufficient to obtain convergence for stochastic gradient descent.

ii. Compared to batch gradient descent, stochastic gradient reaches convergence much more faster. The loss obtained at the end of batch gradient descent was approximately 38. This loss was obtained after 1000 epochs. However, the same loss for stochastic gradient descent was obtained after 2 or 3 epochs.

The accuracy obtained by stochastic gradient descent was after 100 epochs however, for the batch gradient descent it might be over 1000 epochs which is not that feasible.

iii. The learning curve and the accuracy for mini batch of 100 is as follows:





The curves obtained for stochastic gradient descent with a mini batch size of 100 achieves the loss and accuracy that is similar to batch gradient descent in almost 100 epochs compared to 1000 epochs of batch gradient descent. The stochastic gradient descent had a learning rate of 0.01 whereas batch gradient descent had a learning rate of 0.1. If the batch gradient descent had an even higher learning rate then it will achieve convergence faster.

iv. For mini batch size of 100, the updates happen for only 600 datapoints in each epoch. For single point mini batch, 60,000 updates happen in each epoch and for batch gradient descent, the number of updates is dependent on the number of epochs. Single point SGD has provided the fastest convergence with a learning rate of 0.01 whereas for mini batch of 100, the SGD takes more number of epochs to converge as only 600 updates happened in each epoch. For batch gradient descent, the learning rate and the number of epochs decide how fast we can achieve convergence. This is specific to only MNIST dataset. For a different dataset, there might be a scenario where mini batch of 100 SGD or batch gradient might provide better results compared to single point SGD.

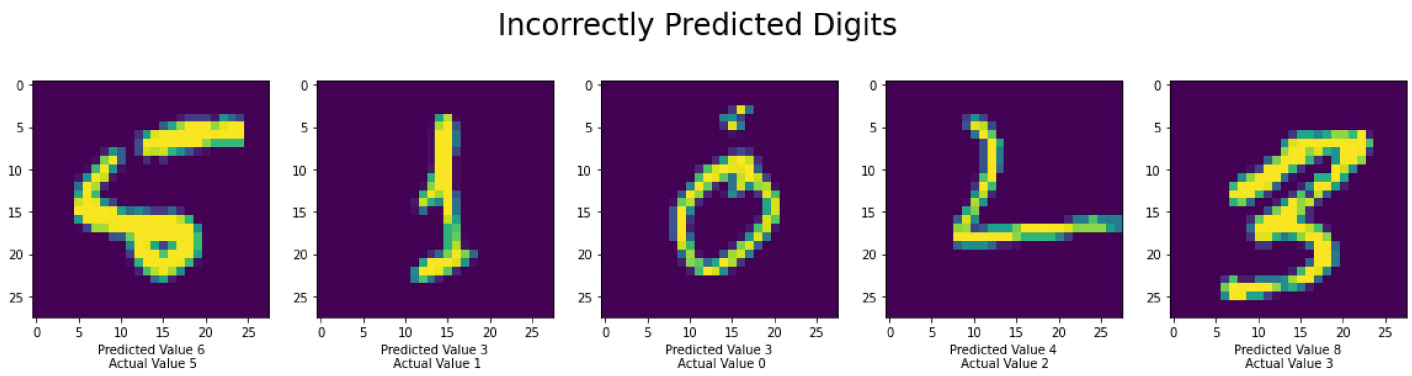
Q2. a) Using the np.asarray function, I extracted each of the weight files into separate numpy arrays. The dimensions of weights was calculated using .shape function. The shape of the weights is a 2D array whereas the biases have a shape of 1D array. The result obtained is shown below:

```
print(w1.shape, w2.shape, w3.shape, b1.shape, b2.shape, b3.shape)
```

```
(200, 784) (100, 200) (10, 100) (200,) (100,) (10,)
```

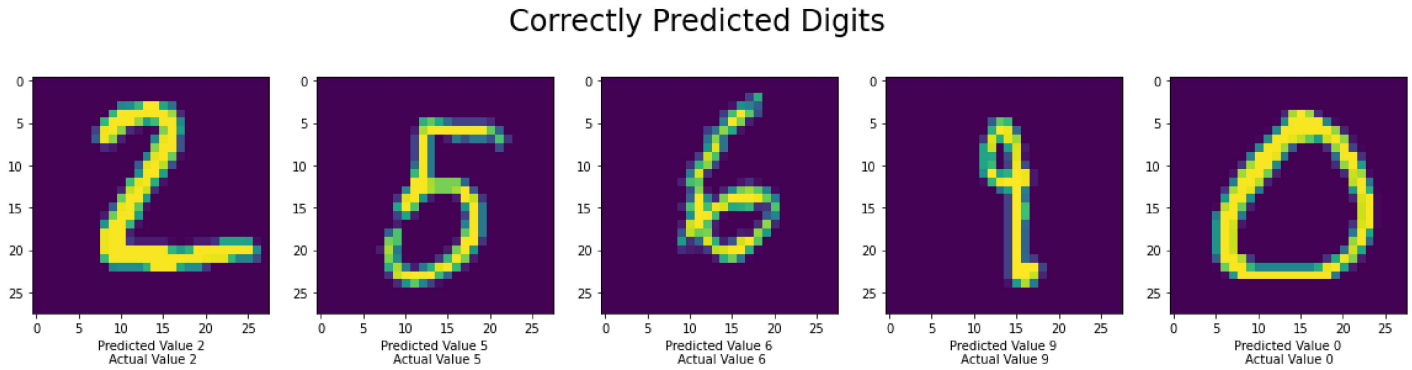
e) After calculating the predicted values of y, I stored these values in a list y1. The true value i.e, ydata was converted into a list and the one-hot encoded vectors were converted into respective numbers. Comparing both of those gave me the result that the out of 10,000 datapoints, my model predicted 9790 datapoints correctly.

f) Out of the 10000 datapoints, the model incorrectly predicted the values of 210 datapoints. I chose 5 datapoints where the model classified the digits incorrectly and 5 datapoints where the model classified the datapoints correctly. The visualizations of the incorrectly predicted digits is as follows:



As seen in the plot, some of the digits that were incorrectly predicted can be predicted properly when inspected visually.Eg: Digits '1' and Digits '2' which were incorrectly predicted as '3' and '4' respectively. However certain digits like '0' and '5' which were incorrectly predicted as '6' and '3', do seem confusing when inspected visually.

The visualisations for the correctly predicted datapoints is as follows:



Appendix

```
#Q1a

import h5py
import numpy as np
import matplotlib.pyplot as plt

data1 = h5py.File('mnist_traindata.hdf5', 'r')
list(data1.keys())

xtrain = np.asarray(data1['xdata'])
ytrain = np.asarray(data1['ydata'])

y1 = ytrain.tolist()

yt = []
for i in range(60000):
  if(y1[i]==[0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]):
    yt.append(1)
  else:
    yt.append(0)
yt = np.array(yt).reshape(-1,1)

data2 = h5py.File('mnist_testdata.hdf5')
xtest = np.asarray(data2['xdata'])
ytest = np.asarray(data2['ydata'])

y2 = []
y2 = ytest.tolist()

ytesto = []
for i in range(10000):
  if(y2[i]==[0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]):
    ytesto.append(1)
  else:
    ytesto.append(0)
```

```
ytesto = np.array(ytesto).reshape(-1,1)

#Intialising the Weights and Bias for the hidden Layer
np.random.seed(seed = 1)
w = np.random.normal(0,10,size = (1,784))
#w = np.zeros((1,784))
#b = np.zeros(1)
b = np.random.randn(1)

#Calculating the predicted values, binary log loss, updating the wieghts and biases
alpha = 0.09
loss = []
epsilon = 1e-40
i = 0
lam = 0.1
loss.append(np.inf)

trainacc = []
testaccu = []

#Appending one iteration of the loss
y = 1/(1 + np.exp(-(np.matmul(xtrain,w.T) + b)))
l = np.sum((np.sum((-yt)*np.log(y + epsilon)-(1-yt)*np.log(1-y + epsilon))))/60000 + lam*np.s
loss.append(l)

losst = []

ytest2 = 1/(1 + np.exp(-(np.matmul(xtest,w.T) + b)))
ltest = np.sum((np.sum((-ytesto)*np.log(ytest2+epsilon)-(1-ytesto)*np.log(1-ytest2+epsilon)))
losst.append(np.inf)
losst.append(ltest)

#for i in range(2500):
while(np.abs(loss[i]-loss[i+1])> 10**-5):

  print( np.sum((y > 0.5) == yt) )
  trainacc.append( np.sum((y > 0.5) == yt) )

  print( np.sum((ytest2 > 0.5) == ytesto) )
  testaccu.append( np.sum((ytest2 > 0.5) == ytesto) )

  dw = (1/60000)*(np.matmul(xtrain.T,(y-yt))) + 2*lam*w.T
  db = (1/60000)*(np.sum(y-yt))
  #print("Shape of db:",db.shape)
  #print(dw)
  #print(db)

  w = w - alpha*dw.T

  #print("Shape of w:",w.shape)
```

```
    b = b - alpha*db
    y = 1/(1 + np.exp(-(np.matmul(xtrain,w.T) + b)))
    #print("Shape of y:",y.shape)
    l = np.sum((np.sum((-yt)*np.log(y+epsilon)-(1-yt)*np.log(1-y+epsilon))))/60000 + lam*np.sum

    #loss for test data
    ytest2 = 1/(1 + np.exp(-(np.matmul(xtest,w.T) + b)))
    ltest = np.sum((np.sum((-ytesto)*np.log(ytest2+epsilon)-(1-ytesto)*np.log(1-ytest2+epsilon)


    #Appending loss to a list
    loss.append(l)
    losst.append(ltest)
    print("Iteration and loss: ", i, l)


    i = i+1

  plt.plot(loss)
  plt.plot(losst)
  plt.xlabel('Number of iterations')
  plt.ylabel('Log Loss')
  plt.title('Log Loss for the model using L2 regularisation, lambda = 0.1 and learning rate = 0
  plt.legend(['Training Set Loss', 'Test Set Loss'])

  #Calculating the Accuracy

  traina = []
  for j in range(len(trainacc)):
    k = trainacc[j]/60000
    traina.append(k)
  testa = []
  for k in range(len(testaccu)):
    l = testaccu[k]/10000
    testa.append(l)

  #Plotting the Graph
  plt.plot(traina)
  plt.plot(testa)
  plt.xlabel('Number of iterations')
  plt.ylabel('Accuracy')
  plt.title('Number of Iterations vs Accuracy')
  plt.legend(['Training Set Accuracy', 'Test Set Accuracy'])

  #Q1b
  import h5py
  import numpy as np
  import matplotlib.pyplot as plt

  data1 = h5py.File('mnist_traindata.hdf5', 'r')
  list(data1.keys())
```

```
xtrain = np.asarray(data1['xdata'])
ytrain = np.asarray(data1['ydata'])
print(xtrain.shape, ytrain.shape)

data2 = h5py.File('mnist_testdata.hdf5')
xtest = np.asarray(data2['xdata'])
ytest = np.asarray(data2['ydata'])

#Initialising the weights
np.random.seed(seed = 0)
w = np.random.normal(0,10, size =(784,10))
b = np.random.randn(10,1)

def softmax(x):
  a = np.exp(x - np.max(x, axis = 1).reshape(-1,1))
  return a/np.sum(a).reshape(-1,1)

def linear(x,w,b):
  return np.matmul(x, w) + b.reshape(1,-1)

def acc(y_pred, y_true):
  z = y_pred >= np.max(y_pred, axis = 1).reshape(-1,1)
  a = np.equal(y_true,z)
  acc = np.sum(np.sum(a, axis = 1)==y_pred.shape[1])
  # count = 0
  # for i in range(y_true.shape[0]):
  #   print(i, y_true.shape[0])
  #   if(np.argmax(y_pred)==np.argmax(y_true)):
  #     count += 1
  return acc/y_true.shape[0]

learning_rate = 0.01
mini_batch = 60000
n_epochs = 1000
n_iters = int(xtrain.shape[0]/mini_batch)
loss_train_list = []
loss_test_list = []

acc_train_list = []
acc_test_list = []

n_updates = 0
for i in range(n_epochs):
  # In each epoch store the loss and y_pred
  for j in range(n_iters):
    if n_updates % 5000 == 0:
      y_train_pred = softmax(linear(xtrain, w, b))
      y_test_pred = softmax(linear(xtest, w, b))
      loss_train = - (1/60000)*np.sum(np.log( np.sum(y_train_pred * ytrain, axis = 1) + 1e-39
      loss_test = - (1/10000)*np.sum(np.log( np.sum(y_test_pred * ytest, axis = 1) + 1e-39))
      acc_train_list.append(acc(y_train_pred, ytrain))
```

```
        acc_test_list.append(acc(y_test_pred, ytest))
        loss_train_list.append(loss_train)
        loss_test_list.append(loss_test)

      n_updates += mini_batch
      x_train_batch = xtrain[j*mini_batch: (j+1)*mini_batch]
      y_train_batch = ytrain[j*mini_batch: (j+1)*mini_batch]
      # print("X train batch shape : {} Y train batch shape : {}".format(x_train_batch.shape, y

      # Perform weight update each time
      y_train_pred_batch = softmax(linear(x_train_batch, w, b))
      dw = (1/mini_batch) * (np.matmul(x_train_batch.T, y_train_batch - y_train_pred_batch))
      # print("DW : ", dw[78])
      db = (1/mini_batch) * (np.sum(y_train_batch - y_train_pred_batch, axis = 0).reshape(-1,1)
      # print("dw shape : {} db shape : {}".format(dw.shape, db.shape))

      w = w + learning_rate * dw
      b = b + learning_rate * db

    # y_train_pred = softmax(linear(xtrain, w, b))
    # y_test_pred = softmax(linear(xtest, w, b))
    # loss_train = - (1/60000)*np.sum(np.log( np.sum(y_train_pred * ytrain, axis = 1) + 1e-39))
    # loss_test = - (1/10000)*np.sum(np.log( np.sum(y_test_pred * ytest, axis = 1) + 1e-39))
    # loss_train_list.append(loss_train)
    # loss_test_list.append(loss_test)
    print("Epoch : ", i)
    print(loss_train)
    print(loss_test)

    import matplotlib.pyplot as plt

    plt.plot(loss_train_list)
plt.plot(loss_test_list)
plt.xlabel('Number of batches')
plt.ylabel('Log Loss')
plt.title('Number of batches vs Log Loss')
plt.legend(['Training Set Loss', 'Test Set Loss'])

plt.plot(acc_train_list)
plt.plot(acc_test_list)
plt.xlabel('Number of batches')
plt.ylabel('Accuracy')
plt.title('Number of batches vs Accuracy')
plt.legend(['Training Set Accuracy', 'Test Set Accuracy'])



#Q1ci.

import h5py
import numpy as np
```

```
import matplotlib.pyplot as plt

data1 = h5py.File('mnist_traindata.hdf5', 'r')
list(data1.keys())

xtrain = np.asarray(data1['xdata'])
ytrain = np.asarray(data1['ydata'])
print(xtrain.shape, ytrain.shape)

data2 = h5py.File('mnist_testdata.hdf5')
xtest = np.asarray(data2['xdata'])
ytest = np.asarray(data2['ydata'])

#Initialising the weights
np.random.seed(seed = 0)
w = np.random.normal(0,10, size =(784,10))
b = np.random.randn(10,1)

def softmax(x):
  a = np.exp(x - np.max(x, axis = 1).reshape(-1,1))
  return a/np.sum(a).reshape(-1,1)

def linear(x,w,b):
  return np.matmul(x, w) + b.reshape(1,-1)

def acc(y_pred, y_true):
  z = y_pred >= np.max(y_pred, axis = 1).reshape(-1,1)
  a = np.equal(y_true,z)
  acc = np.sum(np.sum(a, axis = 1)==y_pred.shape[1])
  # count = 0
  # for i in range(y_true.shape[0]):
  #   print(i, y_true.shape[0])
  #   if(np.argmax(y_pred)==np.argmax(y_true)):
  #     count += 1
  return acc/y_true.shape[0]

learning_rate = 0.01
mini_batch = 1
n_epochs = 100
n_iters = int(xtrain.shape[0]/mini_batch)
loss_train_list = []
loss_test_list = []

acc_train_list = []
acc_test_list = []

n_updates = 0
for i in range(n_epochs):
  # In each epoch store the loss and y_pred
  for j in range(n_iters):
    if n_updates % 5000 == 0:
```

```
        y_train_pred = softmax(linear(xtrain, w, b))
        y_test_pred = softmax(linear(xtest, w, b))
        loss_train = - (1/60000)*np.sum(np.log( np.sum(y_train_pred * ytrain, axis = 1) + 1e-39
        loss_test = - (1/10000)*np.sum(np.log( np.sum(y_test_pred * ytest, axis = 1) + 1e-39))
        acc_train_list.append(acc(y_train_pred, ytrain))
        acc_test_list.append(acc(y_test_pred, ytest))
        loss_train_list.append(loss_train)
        loss_test_list.append(loss_test)

      n_updates += mini_batch
      x_train_batch = xtrain[j*mini_batch: (j+1)*mini_batch]
      y_train_batch = ytrain[j*mini_batch: (j+1)*mini_batch]
      # print("X train batch shape : {} Y train batch shape : {}".format(x_train_batch.shape, y

      # Perform weight update each time
      y_train_pred_batch = softmax(linear(x_train_batch, w, b))
      dw = (1/mini_batch) * (np.matmul(x_train_batch.T, y_train_batch - y_train_pred_batch))
      # print("DW : ", dw[78])
      db = (1/mini_batch) * (np.sum(y_train_batch - y_train_pred_batch, axis = 0).reshape(-1,1)
      # print("dw shape : {} db shape : {}".format(dw.shape, db.shape))

      w = w + learning_rate * dw
      b = b + learning_rate * db

    # y_train_pred = softmax(linear(xtrain, w, b))
    # y_test_pred = softmax(linear(xtest, w, b))
    # loss_train = - (1/60000)*np.sum(np.log( np.sum(y_train_pred * ytrain, axis = 1) + 1e-39))
    # loss_test = - (1/10000)*np.sum(np.log( np.sum(y_test_pred * ytest, axis = 1) + 1e-39))
    # loss_train_list.append(loss_train)
    # loss_test_list.append(loss_test)
    print("Epoch : ", i)
    print(loss_train)
    print(loss_test)

    import matplotlib.pyplot as plt

    plt.plot(loss_train_list)
plt.plot(loss_test_list)
plt.xlabel('Number of batches')
plt.ylabel('Log Loss')
plt.title('Number of batches vs Log Loss')
plt.legend(['Training Set Loss', 'Test Set Loss'])

plt.plot(acc_train_list)
plt.plot(acc_test_list)
plt.xlabel('Number of batches')
plt.ylabel('Accuracy')
plt.title('Number of batches vs Accuracy')
plt.legend(['Training Set Accuracy', 'Test Set Accuracy'])
```

```
#Q1ciii
import h5py
import numpy as np
import matplotlib.pyplot as plt

data1 = h5py.File('mnist_traindata.hdf5', 'r')
list(data1.keys())

xtrain = np.asarray(data1['xdata'])
ytrain = np.asarray(data1['ydata'])
print(xtrain.shape, ytrain.shape)

data2 = h5py.File('mnist_testdata.hdf5')
xtest = np.asarray(data2['xdata'])
ytest = np.asarray(data2['ydata'])

#Initialising the weights
np.random.seed(seed = 0)
w = np.random.normal(0,10, size =(784,10))
b = np.random.randn(10,1)

def softmax(x):
  a = np.exp(x - np.max(x, axis = 1).reshape(-1,1))
  return a/np.sum(a).reshape(-1,1)

def linear(x,w,b):
  return np.matmul(x, w) + b.reshape(1,-1)

def acc(y_pred, y_true):
  z = y_pred >= np.max(y_pred, axis = 1).reshape(-1,1)
  a = np.equal(y_true,z)
  acc = np.sum(np.sum(a, axis = 1)==y_pred.shape[1])
  # count = 0
  # for i in range(y_true.shape[0]):
  #   print(i, y_true.shape[0])
  #   if(np.argmax(y_pred)==np.argmax(y_true)):
  #     count += 1
  return acc/y_true.shape[0]

learning_rate = 0.01
mini_batch = 100
n_epochs = 100
n_iters = int(xtrain.shape[0]/mini_batch)
loss_train_list = []
loss_test_list = []

acc_train_list = []
acc_test_list = []

n_updates = 0
for i in range(n_epochs):
```

```
    # In each epoch store the loss and y_pred
    for j in range(n_iters):
      if n_updates % 5000 == 0:
        y_train_pred = softmax(linear(xtrain, w, b))
        y_test_pred = softmax(linear(xtest, w, b))
        loss_train = - (1/60000)*np.sum(np.log( np.sum(y_train_pred * ytrain, axis = 1) + 1e-39
        loss_test = - (1/10000)*np.sum(np.log( np.sum(y_test_pred * ytest, axis = 1) + 1e-39))
        acc_train_list.append(acc(y_train_pred, ytrain))
        acc_test_list.append(acc(y_test_pred, ytest))
        loss_train_list.append(loss_train)
        loss_test_list.append(loss_test)

      n_updates += mini_batch
      x_train_batch = xtrain[j*mini_batch: (j+1)*mini_batch]
      y_train_batch = ytrain[j*mini_batch: (j+1)*mini_batch]
      # print("X train batch shape : {} Y train batch shape : {}".format(x_train_batch.shape, y_

      # Perform weight update each time
      y_train_pred_batch = softmax(linear(x_train_batch, w, b))
      dw = (1/mini_batch) * (np.matmul(x_train_batch.T, y_train_batch - y_train_pred_batch))
      # print("DW : ", dw[78])
      db = (1/mini_batch) * (np.sum(y_train_batch - y_train_pred_batch, axis = 0).reshape(-1,1)
      # print("dw shape : {} db shape : {}".format(dw.shape, db.shape))

      w = w + learning_rate * dw
      b = b + learning_rate * db

    # y_train_pred = softmax(linear(xtrain, w, b))
    # y_test_pred = softmax(linear(xtest, w, b))
    # loss_train = - (1/60000)*np.sum(np.log( np.sum(y_train_pred * ytrain, axis = 1) + 1e-39))
    # loss_test = - (1/10000)*np.sum(np.log( np.sum(y_test_pred * ytest, axis = 1) + 1e-39))
    # loss_train_list.append(loss_train)
    # loss_test_list.append(loss_test)
    print("Epoch : ", i)
    print(loss_train)
    print(loss_test)

    import matplotlib.pyplot as plt

    plt.plot(loss_train_list)
  plt.plot(loss_test_list)
  plt.xlabel('Number of batches')
  plt.ylabel('Log Loss')
  plt.title('Number of batches vs Log Loss')
  plt.legend(['Training Set Loss', 'Test Set Loss'])

  plt.plot(acc_train_list)
  plt.plot(acc_test_list)
  plt.xlabel('Number of batches')
  plt.ylabel('Accuracy')
  plt.title('Number of batches vs Accuracy')
```

```
plt.legend(['Training Set Accuracy', 'Test Set Accuracy'])
```

```
#Q2

#importing the libraries

import h5py
import numpy as np
import matplotlib.pyplot as plt

#Extracting the weights and biases from .hdf5 file

data1 = h5py.File('mnist_network_params.hdf5', 'r')

#Assigning the weights and biases into separate arrays

w1 = np.asarray(data1['W1'])
w2 = np.asarray(data1['W2'])
w3 = np.asarray(data1['W3'])
b1 = np.asarray(data1['b1'])
b2 = np.asarray(data1['b2'])
b3 = np.asarray(data1['b3'])

#Displaying the shapes of the weights and bias arrays
print(w1.shape, w2.shape, w3.shape, b1.shape, b2.shape, b3.shape)

#Extracting the xdata and ydata from the hdf5 file
inputdata = h5py.File('mnist_testdata.hdf5', 'r')

#Assigning the xdata and ydata into separate arrays and printing the shape
x = np.asarray(inputdata['xdata'])
y = np.asarray(inputdata['ydata'])
print(x.shape, y.shape)

#Defining the ReLU activation function
def relu(x):
  return(np.maximum(x,0))

#Calculating the inputs for hidden layer 1 using ReLU
L1 = np.zeros((10000,200))
L1 = np.matmul(x,w1.T) + b1
relu1 = relu(L1)

#Calculating the inputs for hideen layer 2 using ReLU
L2 = np.zeros((10000,100))
L2 = np.matmul(relu1, w2.T) + b2
relu2 = relu(L2)
```

```
#Calculating the Output
L3 = np.zeros((10000,100))
L3 = np.matmul(relu2, w3.T) + b3

# Defining the Softmax activation function
e = np.exp(L3)
e2 = np.sum(e, axis = 1).reshape(-1,1)
softmax = e/e2

#Writing the outputs of the final layer to json file
l = []
for i in range(10000):
  data = {
        "activations": softmax[i].tolist(),
        "index" : i,
         "classification" : int(softmax[i].argmax()),
  }
  l.append(data)

import json
with open("result.json", "w") as f:
  f.write(json.dumps(l))

ind = np.argmax(softmax, axis = 1)
ind1 = softmax.argmax()
#Comparing the number of correct predictions made by model with ydata
y1 = []
y2 = []
y1 = y.tolist()

#Converting One Hot encoded vectors to digits
for i in range(10000):
  if(y1[i] == [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]):
    y2.append(0)
  elif(y1[i] == [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]):
    y2.append(1)
  elif(y1[i] == [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]):
    y2.append(2)
  elif(y1[i] == [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]):
    y2.append(3)
  elif(y1[i] == [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]):
    y2.append(4)
  elif(y1[i] == [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0]):
    y2.append(5)
  elif(y1[i] == [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]):
    y2.append(6)
  elif(y1[i] == [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0]):
    y2.append(7)
  elif(y1[i] == [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0]):
    y2.append(8)
```

```
    else:
      y2.append(9)

y3 = ind.tolist()

count = 0
diff = []
for a,b in zip(y2,y3):
  if a==b:
    count = count + 1

#Calculating the indexes where the predictions were made incorrectly
for i in range(10000):
  if(y2[i]!=y3[i]):
    diff.append(i)

#Storing the index values of incorrectly predicted digits

d = [8, 900, 7216, 1224, 9944]

#Visualization of incorrectly predicted digits
plt.figure(figsize=[20,20])
i =0

for j in d:
  # for i in range(len(d)):
  plt.subplot(1,len(d), i+1)
  plt.imshow(x[j].reshape(28,28))
  plt.xlabel('Predicted Value {}\n Actual Value {}'.format(y3[j], y2[j]))
  i+=1
plt.suptitle('Incorrectly Predicted Digits', y = 0.6,va = 'center', fontsize = 24)
plt.show()

#Visualization of Correctly Predicted Digits

plt.figure(figsize=[20,20])

t = [1, 45, 2000, 4500, 8567]
i =0
for j in t:
  # for i in range(len(d)):
  plt.subplot(1,len(d), i+1)
  plt.imshow(x[j].reshape(28,28))
  plt.xlabel('Predicted Value {}\n Actual Value {}'.format(y3[j], y2[j]))
  i+=1
plt.suptitle('Correctly Predicted Digits', y = 0.6,va = 'center', fontsize = 24)
plt.show()
```