DEEP LEARNING MODEL ON MNIST DATASET

The architecture of this model comprises of three hidden layer with 500, 250 and 100 being their respective sizes

---

The aim is to model a complete handwritten digit recognizer, train it on the train set comprising 60,000 images and test performance on the test set comprising 10,000 images

---

Baseline model includes:

- **Training data size of MNIST data = 60,000**
  - **with its respective batch size being 64**
- **Testing data size of MNIST data = 10,000**
- **Learning rate = 0.01**
- **number of epochs = 15**

Importing Libraries

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torchvision.datasets as data
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader
from sklearn.metrics import confusion_matrix, classification_report

%matplotlib inline
plt.rcParams['figure.figsize'] = (7.0, 4.0) # set default size of plots

```

Setting up to GPU

In [2]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
device
```

Out[2]: device(type='cuda', index=0)

Importing the MNIST dataset

In [3]:
```
train_set = data.MNIST(root = 'MNIST/raw/train-images-idx3-ubyte', train = True, transform= ToTensor(), down
test_set = data.MNIST(root= 'MNIST/raw/train-images-idx3-ubyte', train= False, transform= ToTensor(), downl
```

Loading the data using DataLoader

In [4]:

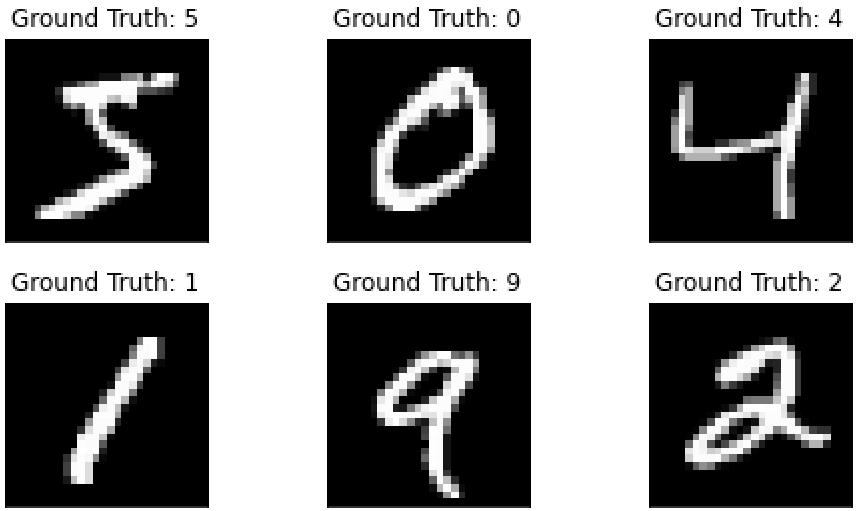
```
train_features = train_set.data #getting images
train_labels = train_set.targets #getting labels
# Visualising data
# Display image and label

print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")

fig = plt.figure()
for i in range(6):
  plt.subplot(2,3,i+1)
  plt.tight_layout()
  plt.imshow(train_features[i].squeeze(), cmap='gray', interpolation='none')
  plt.title("Ground Truth: {}".format(train_labels[i]))
  plt.xticks([])
  plt.yticks([])
plt.show()

```

```
Feature batch shape: torch.Size([60000, 28, 28])
Labels batch shape: torch.Size([60000])
```



One Hot Encoding

In [5]:
```
"""
convert the labels into one-hot format before training and testing the network
"""
def one_hot_encode(Y):
  output = np.eye(10)[np.array(Y).reshape(-1)]
  return output.reshape(list(np.shape(Y))+[10])

```

Activation Functions

Actvation functions and their respective derivatives

(1) Sigmoid

(2) ReLu

(3) TanH

(2) Softmax

In [6]:

```
#ACTIVATION FUNCTIONS
def activation(x, a_type = 'SOFTMAX'):

  if a_type == 'SIGMOID':
    return 1/(1+np.exp(-x))

  if a_type == 'RELU':
    return np.maximum(0,x)

  if a_type == 'TANH':
    return ((np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x)))

  if a_type == 'SOFTMAX':
    return np.exp(x)/sum(np.exp(x))

#DERIVATIVES OF ACTIVATION FUNCTIONS FOR BACK PROPAGATION

def derivative(x, d_type = 'SIGMOID'):

  if d_type == 'SIGMOID':
    s = activation(x, 'SIGMOID')
    return (s*(1-s))

  elif d_type == 'RELU':
    return (x>0)

  elif d_type == 'TANH':
    t = activation(x, 'TANH')
    return (1-t**2)
```

BASELINE

(1) Data Flattening

In [7]:
```
# Flattening and normalising

def data_flattening(features, labels, one_hot = True):
    features = features.numpy()
    labels = labels.numpy()
    X = (features.reshape(features.shape[0], -1)).T
    if one_hot:
      Y = one_hot_encode(labels)
    else:
      Y = labels
    return X, Y
```

(2) Initialising Parameters

In [8]:
```
batchsize = 64
input_size = (train_features.reshape(train_features.shape[0],-1)).shape[1]
hidden_layer_1 = 500
hidden_layer_2 = 250
hidden_layer_3 = 100
output_layer = 10
n_epochs = 15
learning_rate = 0.01
```

In [9]:
```
"""
Function to Initialise the parameters -weights and biases
W1, W2, W3, W4, b1, b2, b3, b4
for the network {i/p -> 500 -> 250 -> 100 ->o/p}
The parameters here are initialised by Xavier initialisation and is one of the common ways of initialisatio
"""
def initialise_parameter(dim):
  np.random.seed(11)

  parameters = {}
  L = len(dim)
  for i in range(1, L):
    Ni = dim[i-1]
    No = dim[i]
    M = np.sqrt(6/(Ni+No))
    parameters["W" + str(i)] = np.asarray(np.random.uniform(-M, M,size = (No,Ni)))
    parameters["b" + str(i)] = np.zeros((dim[i], 1))

    assert(parameters["W" + str(i)].shape == (dim[i], dim[i-1]))
    assert(parameters["b" + str(i)].shape == (dim[i], 1))
  return parameters

```

(3) Forward propagation function

- **using Sigmoid**
- **using TanH**
- **using ReLu**

In [10]:
```
#Function to implement forward propagation
"""
Retrieving the parameters
Linear function on the inputs followed by activation functions

---

In this code we have only got the outputs for Sigmoid and TanH,
for relu seperate code follows

"""
def forward_propagation(X, parameters, forward):
   # retrieve parameters
    W1 = parameters["W1"]
    b1 = parameters["b1"]

    W2 = parameters["W2"]
    b2 = parameters["b2"]

    W3 = parameters["W3"]
    b3 = parameters["b3"]

    W4 = parameters["W4"]
    b4 = parameters["b4"]

    # FORWARD PROPOGATION : LINEAR -> SIGMOID -> LINEAR -> SIGMOID -> LINEAR -> SIGMOID -> LINEAR -> SOFTMA
    # FORWARD PROPOGATION : LINEAR -> TANH -> LINEAR -> TANH -> LINEAR -> TANH -> LINEAR -> SOFTMAX
    # FORWARD PROPOGATION : LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SOFTMAX

    Z1 = np.dot(W1, X) + b1
    A1 =activation(Z1, a_type = forward) # Sigmoid or ReLu or TanH

    Z2 = np.dot(W2, A1) + b2
    A2 = activation(Z2, a_type = forward) # Sigmoid or ReLu or TanH

    Z3 = np.dot(W3, A2) + b3
    A3 = activation(Z3, a_type = forward) # Sigmoid or ReLu or TanH

    Z4 = np.dot(W4, A3) + b4
    A4 = activation(Z4, a_type = 'SOFTMAX') # Softmax

```

```
    cache = {
        "Z1" : Z1, "Z2" : Z2, "Z3" : Z3, "Z4" : Z4,
        "A1" : A1, "A2" : A2, "A3" : A3, "A4" : A4,
        "W1" : W1, "W2" : W2, "W3" : W3, "W4" : W4,
        "b1" : b1, "b2" : b2, "b3" : b3, "b4" : b4}

    return A4, cache

```

(4) Backward Propagation

In [11]:

```python
#Function to implement Backward Propagation
"""
Back Propagation is to get the gradients which will be used
to update the parameters by gradient descent method

"""
def backward_propagation(X, Y, cache, activation):

  m = batchsize
  A4 = cache["A4"]
  A3 = cache["A3"]
  A2 = cache["A2"]
  A1 = cache["A1"]
  Z4 = cache["Z4"]
  Z3 = cache["Z3"]
  Z2 = cache["Z2"]
  Z1 = cache["Z1"]
  W4 = cache["W4"]
  W3 = cache["W3"]
  W2 = cache["W2"]
  W1 = cache["W1"]
  b4 = cache["b4"]
  b3 = cache["b3"]
  b2 = cache["b2"]
  b1 = cache["b1"]

  dZ4 = A4 - Y
  dW4 = 1./m * np.dot(dZ4, A3.T)
  db4 = 1./m * np.sum(dZ4, axis=1, keepdims = True)

  dA3 = np.dot(W4.T, dZ4)
  dZ3 = np.multiply(dA3, derivative(A3, d_type = activation))
  dW3 = 1./m * np.dot(dZ3, A2.T)
  db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)

  dA2 = np.dot(W3.T, dZ3)
  dZ2 = np.multiply(dA2, derivative(A2, d_type = activation))
  dW2 = 1./m * (np.dot(dZ2, A1.T))
  db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

  dA1 = np.dot(W2.T, dZ2)
```

```
    dZ1 = np.multiply(dA1, derivative(A1, d_type = activation))
    dW1 = 1./m * (np.dot(dZ1, X.T))
    db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)

    gradients = {"dW4": dW4, "db4": db4,
                 "dW3": dW3, "db3": db3,
                 "dW2": dW2, "db2": db2,
                 "dW1": dW1, "db1": db1}

    return gradients

```

(5) Update Parameters

In [12]:

```
#Function to update the parameters
"""
Updating the parameters for the next iteration
in order to train the model

---

returns the updated gradients
"""
def update_parameters(parameters, grads, learning_rate):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]
    W4 = parameters["W4"]
    b4 = parameters["b4"]
    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]
    dW3 = grads["dW3"]
    db3 = grads["db3"]
    dW4 = grads["dW4"]
    db4 = grads["db4"]
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    W3 = W3 - learning_rate * dW3
    b3 = b3 - learning_rate * db3
    W4 = W4 - learning_rate * dW4
    b4 = b4 - learning_rate * db4

    parameters={"W1":W1, "b1":b1,
                "W2":W2, "b2":b2,
                "W3":W3, "b3":b3,
                "W4":W4, "b4":b4}
    return parameters
```

(6) Compute Cost

In [13]:

```
"""
Function to compute cross entropy cost
---
Cross Entropy Loss is used to compute the cost in case of classification models
"""
def compute_cost(A, Y):
  m = Y.shape[1]
  logprobs = np.multiply(-np.log(A),Y) + np.multiply(-np.log(1 - A), (1 - Y))
  cost = 1./m * np.nansum(logprobs)
  return cost
```

(8) Predict Labels

In [14]:

```
# Predict
def predict(X, Y, parameters, batchsize = batchsize):

    m = batchsize
    y = Y.T
    p = np.zeros((m), dtype = np.int)
    y = y.numpy()
    # Forward propagation
    a4, caches = forward_propagation(X, parameters)
    print(a4)
    p = np.argmax(a4, axis = 0)
    # print results
    print(f"predicted_value is {p}")
    print(f"label value is {y}")
    print("Accuracy: "  + str(np.mean((p == y))))
    return p
```

(9) Accuracy

In [15]:
```python
"""
Computing the accuracy of the testing and training sets

---
returns the actual and predicted value of y
prints the accuracy

"""

def Accuracy(dataset, parameters, forward, size):

    features = dataset.data
    labels = dataset.targets
    X, Y = data_flattening(features, labels, one_hot = False)
    y = Y.T
    p = np.zeros(size, dtype = int)
    # Forward propagation
    a4, caches = forward_propagation(X, parameters, forward)
    p = np.argmax(a4, axis = 0)
    a = np.mean((p == y))
    print("accuracy is =" + str(a))
    return y, p
```

(10) Gradient Descent Model

In [24]:

```
"""
Model to implement Neural Network by Gradient Descent
---
This function will run once for the entire dataset
---
Input parameters are - train_loader, parameters, type of activation, Learning rate, iterations and lamda
---
This function trains model for both regularised and unregularised

"""
def gradient_descent(dataloader, parameters, forward, learning_rate = 0.01, number_of_iterations = int(6000

    grads = {}
    costs = []
    cost_out = []
    m = batchsize  #number of examples
    for i in range(0, number_of_iterations):
        features, labels = next(iter(dataloader))
        X, Y = data_flattening(features, labels, True)
        a4, cache = forward_propagation(X, parameters, forward)   # FORWARD PROPOGATION : LINEAR -> SIGMOID
        # Cost function
        if lambd == 0:
            cost = compute_cost(a4, Y.T)
            grads = backward_propagation(X, Y.T, cache, forward)
        else:
            cost = compute_cost_with_regularization(a4, Y.T, parameters, lambd)
            grads = backward_propagation_with_regularization(X, Y.T, cache, lambd, forward)

        parameters = update_parameters(parameters, grads, learning_rate)   # Update parameters
        costs.append(cost)
        if i%200==0 or i==(number_of_iterations-1):
                print("Cost after iteration {}: {}".format(i, cost))
                #plot the cost
                plt.plot(costs)
                plt.ylabel('cost')
                plt.xlabel('iterations')
                plt.title("Learning rate =" + str(learning_rate))
                plt.show()
                costs = []
        if i%500 == 0:
            cost_out.append(cost)
```

```
                acc = []
                # Load images to a Torch Variable
                images = train_set.data
                labels_1 = train_set.targets
                X, Y = data_flattening(images, labels_1, one_hot = False)
                y = Y.T
                predicted = np.zeros(60000, dtype = int)
                # Forward propagation
                a4, caches = forward_propagation(X, parameters, forward)
                predicted = np.argmax(a4, axis = 0)
                accuracy = np.mean((predicted == y))
                # Print Loss
                acc.append(accuracy)

    return parameters, cost_out, acc

```

Neural Network

In [27]:

```
"""
Neural Network Function
---
Calls the model for the epoch number of times
"""

def NeuralNet(dataset, forward = 'SIGMOID', learning_rate = 0.01, n_epochs = 1, batch_size = batchsize, lamb
    costs = []
    accuracy = []
    m = batch_size #number of examples
    layers_dim = [input_size, hidden_layer_1, hidden_layer_2, hidden_layer_3, output_layer]
    parameters = initialise_parameter(layers_dim)
    for epoch in range(n_epochs):
        print(f"Epoch :- {epoch+1}")
        train_dataloader = DataLoader(train_set, batch_size=batchsize, shuffle=True)
        parameters, cost, acc = gradient_descent(train_dataloader, parameters, forward, learning_rate = lea
        print (f'Epoch [{epoch+1}/{n_epochs}], Cost: {cost[-1]}, Accuracy: {acc[-1]}')
        accuracy.append(acc)
        costs.append(cost)
    #plot the cost
    plt.plot(costs)
    plt.ylabel('cost')
    plt.xlabel('epochs')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()
    return parameters, accuracy
```

Training

Sigmoid Activation Function

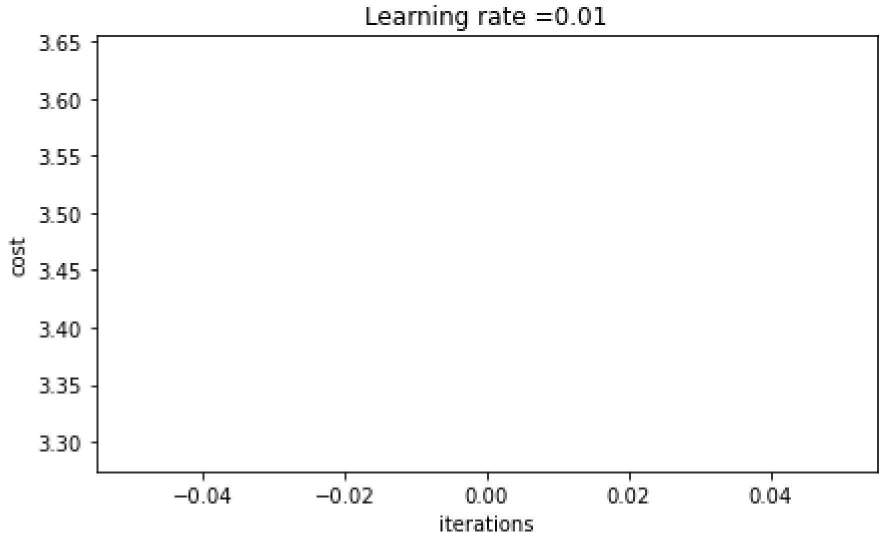
In [48]:

```
"""
Training the Neural Network
Getting trained parameters and accuracy of train dataset

"""
parameters, train_acc = NeuralNet(train_set, forward = 'SIGMOID', learning_rate = 0.01, n_epochs = 15,batch_

```

```
Epoch :- 1
Cost after iteration 0: 3.4647339306614118
```



In [49]:

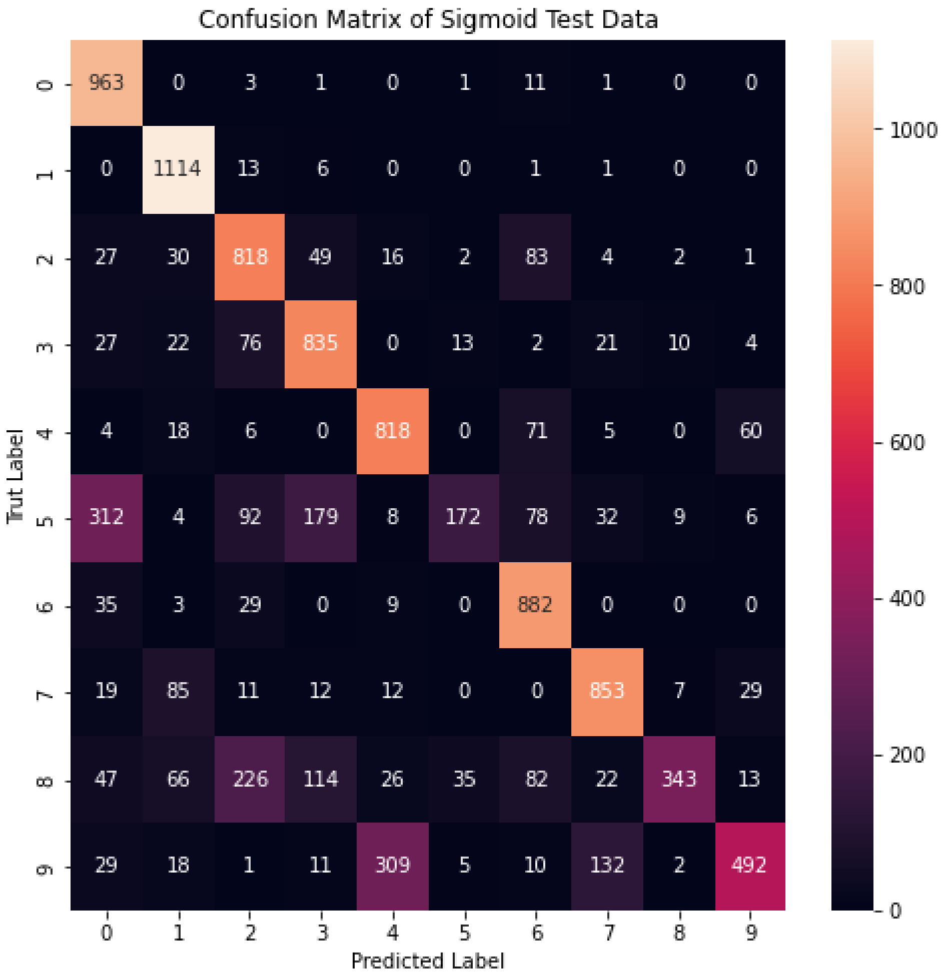
```
print ("On the TEST set:")
y_actual_test, y_pred_test = Accuracy(test_set, parameters,'SIGMOID', 10000)
```

```
On the TEST set:
accuracy is =0.729
```

```
/home/mansi/.conda/envs/hbp/lib/python3.7/site-packages/ipykernel_launcher.py:5: RuntimeWarning: overflow enc
ountered in exp
  """
```

In [52]:
```
#to get the heatmap for the confusion matrix
import seaborn as sn
plt.figure(figsize=(8,8))
sn.heatmap(confusion_matrix(y_actual_test, y_pred_test),annot=True,fmt='d')
plt.xlabel('Predicted Label')
plt.ylabel('Trut Label')
plt.title('Confusion Matrix of Sigmoid Test Data')
```

Out[52]:
```
Text(0.5, 1.0, 'Confusion Matrix of Sigmoid Test Data')
```



In [51]:
```
c = confusion_matrix(y_actual_test, y_pred_test)
print(c)
```

```
[[ 963    0    3    1    0    1   11    1    0    0]
 [   0 1114   13    6    0    0    1    1    0    0]
 [  27   30  818   49   16    2   83    4    2    1]
 [  27   22   76  835    0   13    2   21   10    4]
 [   4   18    6    0  818    0   71    5    0   60]
 [ 312    4   92  179    8  172   78   32    9    6]
 [  35    3   29    0    9    0  882    0    0    0]
 [  19   85   11   12   12    0    0  853    7   29]
 [  47   66  226  114   26   35   82   22  343   13]
 [  29   18    1   11  309    5   10  132    2  492]]
```

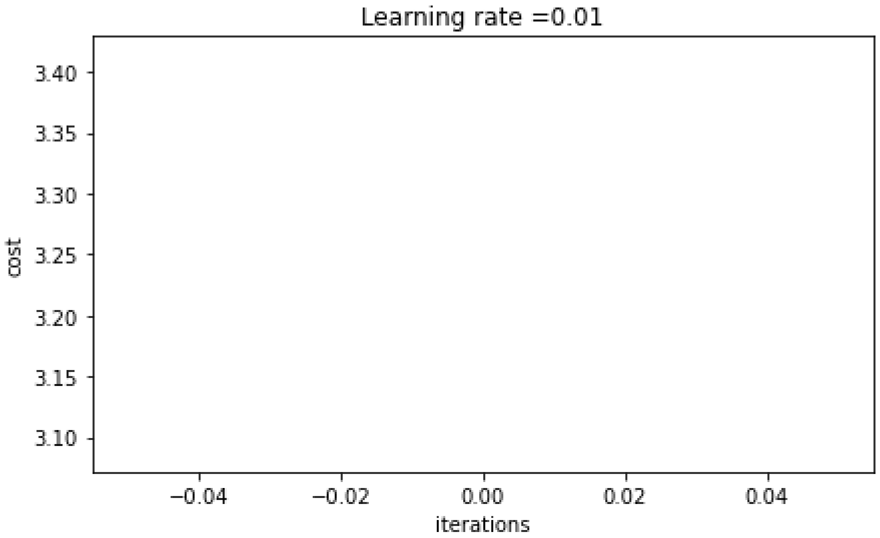
Observations:

- **For $\eta = 0.01$, batch size = 64 and epochs = 15**
  - Train Accuracy = 71%
  - Test Accuracy = 72.9%
- **For $\eta = 0.04$, batch size = 64 and epochs = 15**
  - Train Accuracy = 85.33%
  - Test Accuracy = 85.77%
- **For $\eta = 0.01$, batch size = 64 and epochs = 10**
  - Train Accuracy = 89.33%
  - Test Accuracy = 88.77%

TanH activation function

In [71]:
```
parameters_tanh, accuracy = NeuralNet(train_set, forward = 'TANH', learning_rate = 0.01, n_epochs = 10,batcl
```

```
Epoch :- 1
Cost after iteration 0: 3.2504734281645433
```



In [72]:
```
print ("On the TEST set:")
y_actual_test, y_pred_test = Accuracy(test_set, parameters_tanh, 'TANH', 10000)
```

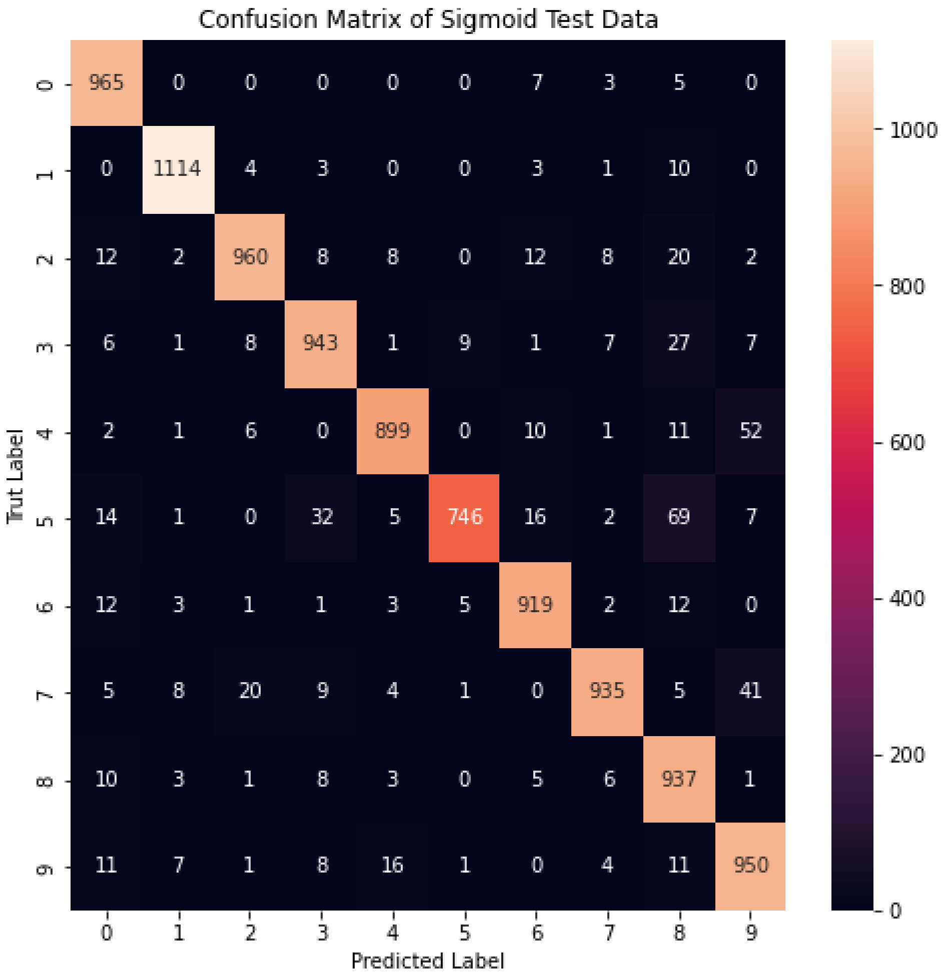
```
On the TEST set:

/home/mansi/.conda/envs/hbp/lib/python3.7/site-packages/ipykernel_launcher.py:11: RuntimeWarning: overflow en
countered in exp
  # This is added back by InteractiveShellApp.init_path()
/home/mansi/.conda/envs/hbp/lib/python3.7/site-packages/ipykernel_launcher.py:11: RuntimeWarning: invalid val
ue encountered in true_divide
  # This is added back by InteractiveShellApp.init_path()

accuracy is =0.9368
```

In [73]:
```python
#to get the heatmap for the confusion matrix
import seaborn as sn
plt.figure(figsize=(8,8))
sn.heatmap(confusion_matrix(y_actual_test, y_pred_test),annot=True,fmt='d')
plt.xlabel('Predicted Label')
plt.ylabel('Trut Label')
plt.title('Confusion Matrix of Sigmoid Test Data')
```

Out[73]:
```
Text(0.5, 1.0, 'Confusion Matrix of Sigmoid Test Data')
```



In [74]:
```
c = confusion_matrix(y_actual_test, y_pred_test)
print(c)
```

```
[[ 965    0    0    0    0    0    7    3    5    0]
 [   0 1114    4    3    0    0    3    1   10    0]
 [  12    2  960    8    8    0   12    8   20    2]
 [   6    1    8  943    1    9    1    7   27    7]
 [   2    1    6    0  899    0   10    1   11   52]
 [  14    1    0   32    5  746   16    2   69    7]
 [  12    3    1    1    3    5  919    2   12    0]
 [   5    8   20    9    4    1    0  935    5   41]
 [  10    3    1    8    3    0    5    6  937    1]
 [  11    7    1    8   16    1    0    4   11  950]]
```

Observations:

- **For $\eta = 0.01$, batch size = 64 and epochs = 15**
  - Train Accuracy = 94.39%
  - Test Accuracy = 93.92%
- **For $\eta = 0.04$, batch size = 64 and epochs = 15**
  - Train Accuracy = 85.66%
  - Test Accuracy = 85.25%
- **For $\eta = 0.01$, batch size = 64 and epochs = 10**
  - Train Accuracy = 93.7%
  - Test Accuracy = 93.68%

Overall Observations:

- Model gives good accuracy in all the three activation functions with their respective best parameters. But the Pytorch model was much faster and still gives better accuracy than all the three here.
- ReLU and TanH give better accuracy as compared to Sigmoid

- ReLU and tanh give better accuracy as compared to sigmoid
- After applying regularisation we donot observe much change in accuracy, but overall performance becomes better

- When we observe PyTorch Model,
    - we can tell that tanh doesnot perform as good as sigmoid and relu
    - but in our model tanh gives good results, probably tanh works good with gradient descent than with adam optimiser
    - ReLu shows the best results
    - overall the accuracy is high

- Uniform random Initialisation gave better results than random initialisation or zero initialisation