In this assignment, you'll implement a L-layer deep model on MNIST dataset using Keras. The MNIST dataset contains tens of thousands of scanned images of handwritten digits, together with their correct classifications. MNIST's name comes from the fact that it is a modified subset of two data sets collected by NIST, the United States' National Institute of Standards and Technology.

To use Keras, you'll need to install Keras and Tensorflow.
Please run the following commands if you don't have Keras and TensorFlow already installed.

1. ! pip install TensorFlow
2. ! pip install keras
3. ! pip install msgpack

In [1]:
```
import numpy as np
import pickle
import gzip
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import h5py
import sklearn
import sklearn.datasets
import scipy
import tensorflow as tf

from PIL import Image
from scipy import ndimage


from keras.models import Sequential
from keras.layers import Dense, Dropout, BatchNormalization, Activation
from keras import regularizers

np.random.seed(7)


%matplotlib inline
```

The MNIST dataset we use here is 'mnist.pkl.gz' which is divided into training, validation and test data. The following function *load_data()* unpacks the file and extracts the training, validation and test data.

In [2]:
```
def load_data():
    f = gzip.open('mnist.pkl.gz', 'rb')
    f.seek(0)
    training_data, validation_data, test_data = pickle.load(f, encoding='latin
1')
    f.close()
    return (training_data, validation_data, test_data)
```

Let's see how the data looks:

In [3]:
```
training_data, validation_data, test_data = load_data()
```

In [4]:
```
training_data
```

Out[4]:
```
(array([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]], dtype=float32),
 array([5, 0, 4, ..., 8, 4, 8], dtype=int64))
```

In [5]:
```
print("The feature dataset is:" + str(training_data[0]))
print("The target dataset is:" + str(training_data[1]))
print("The number of examples in the training dataset is:" + str(len(training_
data[0])))
print("The number of points in a single input is:" + str(len(training_data[0][
1])))
```

```
The feature dataset is:[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
The target dataset is:[5 0 4 ... 8 4 8]
The number of examples in the training dataset is:50000
The number of points in a single input is:784
```

Now, as discussed earlier in the lectures, the target variable is converted to a one hot matrix. We use the function *one_hot* to convert the target dataset to one hot encoding.

In [6]:
```
def one_hot(j):
    # input is the target dataset of shape (1, m) where m is the number of dat
a points
    # returns a 2 dimensional array of shape (10, m) where each target value i
s converted to a one hot encoding
    # Look at the next block of code for a better understanding of one hot enc
oding
    n = j.shape[0]
    new_array = np.zeros((10, n))
    index = 0
    for res in j:
        new_array[res][index] = 1.0
        index = index + 1
    return new_array
```

In [7]:
```
data = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
one_hot(data)
```

Out[7]:
```
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

In [8]:
```
def data_wrapper():
    tr_d, va_d, te_d = load_data()

    training_inputs = np.array(tr_d[0][:]).T
    training_results = np.array(tr_d[1][:])
    train_set_y = one_hot(training_results)

    validation_inputs = np.array(va_d[0][:]).T
    validation_results = np.array(va_d[1][:])
    validation_set_y = one_hot(validation_results)

    test_inputs = np.array(te_d[0][:]).T
    test_results = np.array(te_d[1][:])
    test_set_y = one_hot(test_results)

    return (training_inputs, train_set_y, validation_inputs, validation_set_y)
```

In [9]:
```
train_set_x, train_set_y, test_set_x, test_set_y = data_wrapper()
```

For implementing in Keras, the input training and input target dataset are supposed to have shape (m, n) where m is the number of training samples and n is the number of parts in a single input.
Hence, let create the desired dataset shapes by taking transpose.

In [10]:
```
train_set_x = train_set_x.T
train_set_y = train_set_y.T
test_set_x = test_set_x.T
test_set_y = test_set_y.T
```

Now, let's see if the datasets are in the desired shape:

In [11]:

```
print ("train_set_x shape: " + str(train_set_x.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x shape: " + str(test_set_x.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

```
train_set_x shape: (50000, 784)
train_set_y shape: (50000, 10)
test_set_x shape: (10000, 784)
test_set_y shape: (10000, 10)
```

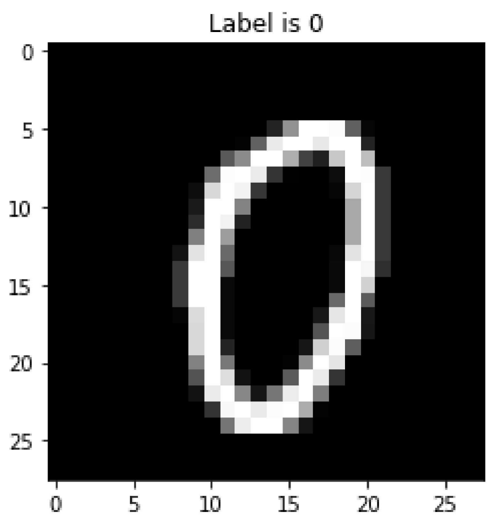
Now let us visualise the dataset. Feel free to change the index to see if the training data has been correctly tagged.

In [12]:

```
index  = 1000
k = train_set_x[index,:]
k = k.reshape((28, 28))
plt.title('Label is {label}'.format(label= training_data[1][index]))
plt.imshow(k, cmap='gray')
```

Out[12]:

```
<matplotlib.image.AxesImage at 0x28d612e28d0>
```



Keras is a framework. So, to implement a neural network model in Keras, we first create an instance of Sequential().
The Sequential model is a linear stack of layers. We then keep adding Dense layers that are fully connected layers as we desire.

We have included Dropout using *nn_model.add(Dropout(0.3))*

We can also include regularization using the command
nn_model.add(Dense(21, activation='relu', kernel_regularizer=regularizers.l2(0.01)))
instead of
nn_model.add(Dense(21, activation='relu'))

In [13]:
```
# create model
nn_model = Sequential()
nn_model.add(Dense(35, input_dim=784, activation='relu'))
nn_model.add(Dropout(0.3))
nn_model.add(Dense(21, activation = 'relu'))
nn_model.add(Dense(10, activation='softmax'))
```

Before we run the model on the training datasets, we compile the model in which we define various things like the loss function, the optimizer and the evaluation metric.

In [14]:
```
nn_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=[
'accuracy'])
```

Now, to fit the model on the training input and training target dataset, we run the following command using a minibatch of size 10 and 10 epochs.

In [15]:
```
nn_model.fit(train_set_x, train_set_y, epochs=10, batch_size=10)
```

```
Epoch 1/10
5000/5000 [==============================] - 10s 2ms/step - loss: 0.5139 - ac
curacy: 0.8412
Epoch 2/10
5000/5000 [==============================] - 10s 2ms/step - loss: 0.3258 - ac
curacy: 0.8993
Epoch 3/10
5000/5000 [==============================] - 10s 2ms/step - loss: 0.2944 - ac
curacy: 0.9068
Epoch 4/10
5000/5000 [==============================] - 9s 2ms/step - loss: 0.2760 - acc
uracy: 0.9131
Epoch 5/10
5000/5000 [==============================] - 10s 2ms/step - loss: 0.2613 - ac
curacy: 0.9184
Epoch 6/10
5000/5000 [==============================] - 9s 2ms/step - loss: 0.2535 - acc
uracy: 0.9200
Epoch 7/10
5000/5000 [==============================] - 9s 2ms/step - loss: 0.2419 - acc
uracy: 0.9234
Epoch 8/10
5000/5000 [==============================] - 10s 2ms/step - loss: 0.2308 - ac
curacy: 0.9264
Epoch 9/10
5000/5000 [==============================] - 10s 2ms/step - loss: 0.2328 - ac
curacy: 0.9264
Epoch 10/10
5000/5000 [==============================] - 10s 2ms/step - loss: 0.2276 - ac
curacy: 0.9283
```

Out[15]:
```
<tensorflow.python.keras.callbacks.History at 0x28d62725da0>
```

In [16]:
```
scores_train = nn_model.evaluate(train_set_x, train_set_y)
print("\n%s: %.2f%%" % (nn_model.metrics_names[1], scores_train[1]*100))
```

```
1563/1563 [==============================] - 3s 2ms/step - loss: 0.1023 - acc
uracy: 0.9687

accuracy: 96.87%
```

We can see that the model has ~ 97% accuracy on the training dataset.

Now, let's make predictions on the test dataset.

In [17]:
```
predictions = nn_model.predict(test_set_x)
predictions = np.argmax(predictions, axis = 1)
predictions
```

Out[17]:
```
array([3, 8, 6, ..., 5, 6, 8], dtype=int64)
```

In [18]:
```
scores_test = nn_model.evaluate(test_set_x, test_set_y)
print("\n%s: %.2f%%" % (nn_model.metrics_names[1], scores_test[1]*100))
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1330 - accur
acy: 0.9627

accuracy: 96.27%
```

We can see that the model has ~96% accuracy on the training dataset.

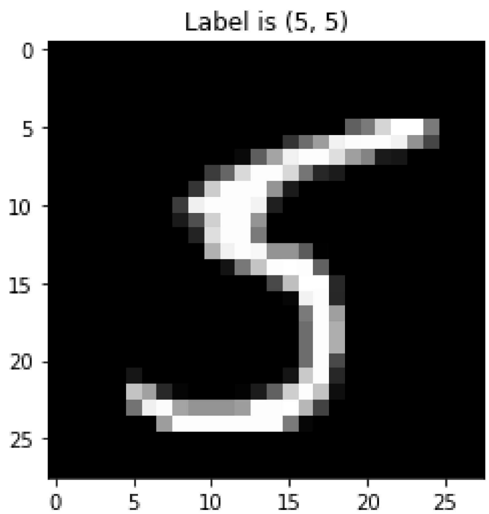
Try and look at the different test cases and check which all have gone wrong. Feel free to change the index numbers.

In [19]:

```
index  = 9997
k = test_set_x[index, :]
k = k.reshape((28, 28))
plt.title('Label is {label}'.format(label=(predictions[index], np.argmax(test_set_y, axis = 1)[index])))
plt.imshow(k, cmap='gray')
```

Out[19]:

```
<matplotlib.image.AxesImage at 0x28d628ec470>
```



In [20]:

```
index  = 6897
k = test_set_x[index, :]
k = k.reshape((28, 28))
plt.title('Label is {label}'.format(label=(predictions[index], np.argmax(test_set_y, axis = 1)[index])))
plt.imshow(k, cmap='gray')
```

Out[20]:

```
<matplotlib.image.AxesImage at 0x28d6292da90>
```