In [2]:
```
'''
Implementing Autoencoders in Keras

Autoencoders are similar to dimensionality reduction techniques like Principal Component Analysis (PCA).
They project the data from a higher dimension to a lower dimension using linear transformation and
try to preserve the important features of the data while removing the non-essential parts.

PCA uses linear transformation whereas

autoencoders use non-linear transformations

Dataset Used: NotMNIST -images of font glypyhs for the letters A through J.

'''
'''
The OS module in Python provides a way of using operating system dependent functionality
The functions that the OS module provides allows you to interface with the underlying operating system that
Python is running on – be that Windows, Mac or Linux
'''

import os
```

In [3]:
```
#All imports
'''
A function that opens the gzip file, reads the file using bytestream.read()

'''
import keras
from matplotlib import pyplot as plt
import numpy as np
import gzip
%matplotlib inline
from keras.layers import Input,Conv2D,MaxPooling2D,UpSampling2D
from keras.models import Model
from keras.optimizers import RMSprop
```

Using TensorFlow backend.

In [6]:
```
'''
Pass the image dimension and the total number of images to this function

using np.frombuffer(), you convert the string stored in variable buf
into a NumPy array of type float32

Reshape the array into a three-dimensional array or tensor
where the first dimension is number of images,
and the second and third dimension being the dimension of the image.

Finally, return the NumPy array data
'''

def extract_data(filename, num_images):
    with gzip.open(filename) as bytestream:
        bytestream.read(16)
        buf = bytestream.read(28 * 28 * num_images)
        data = np.frombuffer(buf, dtype=np.uint8).astype(np.float32)
        data = data.reshape(num_images, 28,28)
        return data
```

In [8]:
```
'''
call the function extract_data() by passing

the training and testing files along with their corresponding number of images

'''
train_data = extract_data('C:/MLCourse/notMNIST-to-MNIST-master/train-images-idx3-ubyte.gz', 60000)
test_data = extract_data('C:/MLCourse/notMNIST-to-MNIST-master/t10k-images-idx3-ubyte.gz', 10000)
```

In [9]:
```
def extract_labels(filename, num_images):
    with gzip.open(filename) as bytestream:
        bytestream.read(8)
        buf = bytestream.read(1 * num_images)
        labels = np.frombuffer(buf, dtype=np.uint8).astype(np.int64)
        return labels
```

In [10]:
```
train_labels = extract_labels('C:/MLCourse/notMNIST-to-MNIST-master/train-labels-idx1-ubyte.gz',60000)
test_labels = extract_labels('C:/MLCourse/notMNIST-to-MNIST-master/t10k-labels-idx1-ubyte.gz',10000)
```

In [11]:
```
'''
analyze how images in the dataset look like and also see the dimension of the images
'''
# Shapes of training set
print("Training set (images) shape: {shape}".format(shape=train_data.shape))
```

```
Training set (images) shape: (60000, 28, 28)
```

In [12]:
```
# Shapes of test set
print("Test set (images) shape: {shape}".format(shape=test_data.shape))
```

```
Test set (images) shape: (10000, 28, 28)
```

In [13]:
```
# Create dictionary of target classes
#create a dictionary that will have class names with their corresponding categorical class labels
label_dict = {
 0: 'A',
 1: 'B',
 2: 'C',
 3: 'D',
 4: 'E',
 5: 'F',
 6: 'G',
 7: 'H',
 8: 'I',
 9: 'J',
}
```

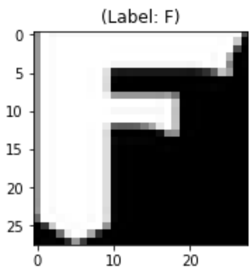
In [14]:
```
plt.figure(figsize=[5,5])
```

Out[14]:
```
<Figure size 360x360 with 0 Axes>
```

```
<Figure size 360x360 with 0 Axes>
```

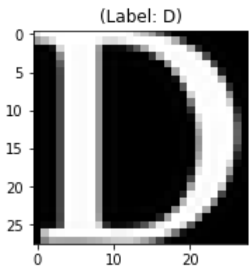
In [15]:
```
# Display the first image in training data
plt.subplot(121)
curr_img = np.reshape(train_data[0], (28,28))
curr_lbl = train_labels[0]
plt.imshow(curr_img, cmap='gray')
plt.title("(Label: " + str(label_dict[curr_lbl]) + ")")
```

Out[15]:
```
Text(0.5, 1.0, '(Label: F)')
```



In [16]:
```
# Display the first image in testing data
plt.subplot(122)
curr_img = np.reshape(test_data[0], (28,28))
curr_lbl = test_labels[0]
plt.imshow(curr_img, cmap='gray')
plt.title("(Label: " + str(label_dict[curr_lbl]) + ")")
```

Out[16]:
```
Text(0.5, 1.0, '(Label: D)')
```



In [17]:
```
#Data Preprocessing
'''
first convert each 28 x 28 image of train and test set

into a matrix of size 28 x 28 x 1,

which you can feed into the network

'''

train_data = train_data.reshape(-1, 28,28, 1)
test_data = test_data.reshape(-1, 28,28, 1)
train_data.shape, test_data.shape
```

Out[17]:
```
((60000, 28, 28, 1), (10000, 28, 28, 1))
```

In [18]:
```
'''

make sure to check the data type of the training and testing NumPy arrays,

it should be in float32 format, if not you will need to convert it into this format
'''
train_data.dtype, test_data.dtype
```

Out[18]:
```
(dtype('float32'), dtype('float32'))
```

In [19]:
```
'''
rescale the training and testing data with the

maximum pixel value of the training and testing data

Maximum pixel value was 255

'''
np.max(train_data), np.max(test_data)
```

Out[19]:
```
(255.0, 255.0)
```

In [20]:
```
train_data = train_data / np.max(train_data)
test_data = test_data / np.max(test_data)
```

In [21]:
```
'''
verify the maximum value of training and testing data

which should be 1.0 after rescaling it

'''

np.max(train_data), np.max(test_data)
```

Out[21]:
```
(1.0, 1.0)
```

In [22]:
```
'''
train the model on 80% of the data and validate it on 20% of the remaining training data
'''
from sklearn.model_selection import train_test_split
train_X,valid_X,train_ground,valid_ground = train_test_split(train_data,
                                                             train_data,
                                                             test_size=0.2,
                                                             random_state=13)

'''
We don't need training and testing labels.
Why ??? Because we will pass the training images twice.
Training images will both act as the input as well as the ground truth
similar to the labels you have in classification task.

'''
```

In [ ]:
```
#The Convolutional Autoencoder

'''
The images are of size 28 x 28 x 1 or a 784-dimensional vector

Convert the image matrix to an array, rescale it between 0 and 1,

reshape it so that it's of size 28 x 28 x 1, and

feed this as an input to the network

'''
```

In [24]:
```
'''
will use a batch size of 128

using a higher batch size of 256 or 512 is also preferable

it all depends on the system you train your model

It contributes heavily in determining the learning parameters and affects the prediction accuracy.

train your network for 50 epochs.
'''

batch_size = 128
epochs = 50
inChannel = 1
x, y = 28, 28
input_img = Input(shape = (x, y, inChannel))
```

In [ ]:
```
'''
Autoencoder is divided into two parts: there's an encoder and a decoder

Encoder

The first layer will have 32 filters of size 3 x 3, followed by a downsampling (max-pooling) layer,
The second layer will have 64 filters of size 3 x 3, followed by another downsampling layer,
The final layer of encoder will have 128 filters of size 3 x 3.

Decoder
The first layer will have 128 filters of size 3 x 3 followed by a upsampling layer,/li>
The second layer will have 64 filters of size 3 x 3 followed by another upsampling layer,
The final layer of encoder will have 1 filter of size 3 x 3.

'''
```

In [25]:
```
# Defining the autoencoder module

def autoencoder(input_img):
    #encoder
    #input = 28 x 28 x 1 (wide and thin)
    conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img) #28 x 28 x 32
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1) #14 x 14 x 32
    conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(pool1) #14 x 14 x 64
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2) #7 x 7 x 64
    conv3 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool2) #7 x 7 x 128 (small and thick)

    #decoder
    conv4 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv3) #7 x 7 x 128
    up1 = UpSampling2D((2,2))(conv4) # 14 x 14 x 128
    conv5 = Conv2D(64, (3, 3), activation='relu', padding='same')(up1) # 14 x 14 x 64
    up2 = UpSampling2D((2,2))(conv5) # 28 x 28 x 64
    decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(up2) # 28 x 28 x 1
    return decoded
```

In [29]:
```
'''
After the model is created,

compile it using the optimizer to be RMSProp.
'''
autoencoder = Model(input_img, autoencoder(input_img))
autoencoder.compile(loss='mean_squared_error', optimizer = RMSprop())
```

In [30]:
```
#visualize the layers created
autoencoder.summary()
```

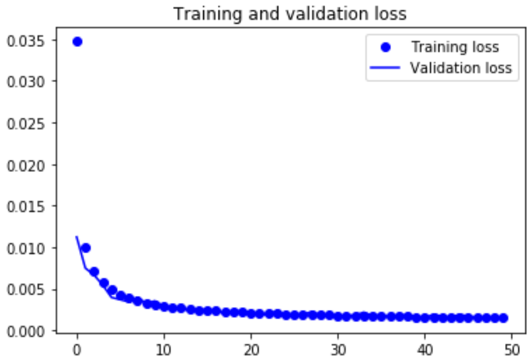
```
Model: "model_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
model_2 (Model)              (None, 28, 28, 1)         314625
=================================================================
Total params: 314,625
Trainable params: 314,625
Non-trainable params: 0
_________________________________________________________________
```

In [31]:
```
autoencoder_train = autoencoder.fit(train_X, train_ground, batch_size=batch_size,epochs=epochs,verbose=1,validation
_data=(valid_X, valid_ground))
```

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/50
48000/48000 [==============================] - 358s 7ms/step - loss: 0.0347 - val_loss: 0.0112
Epoch 2/50
48000/48000 [==============================] - 256s 5ms/step - loss: 0.0100 - val_loss: 0.0075
Epoch 3/50
48000/48000 [==============================] - 187s 4ms/step - loss: 0.0071 - val_loss: 0.0067
Epoch 4/50
48000/48000 [==============================] - 187s 4ms/step - loss: 0.0058 - val_loss: 0.0054
Epoch 5/50
48000/48000 [==============================] - 187s 4ms/step - loss: 0.0049 - val_loss: 0.0040
Epoch 6/50
48000/48000 [==============================] - 187s 4ms/step - loss: 0.0043 - val_loss: 0.0037
Epoch 7/50
48000/48000 [==============================] - 188s 4ms/step - loss: 0.0039 - val_loss: 0.0034
Epoch 8/50
48000/48000 [==============================] - 187s 4ms/step - loss: 0.0036 - val_loss: 0.0040
Epoch 9/50
48000/48000 [==============================] - 188s 4ms/step - loss: 0.0033 - val_loss: 0.0033
Epoch 10/50
48000/48000 [==============================] - 188s 4ms/step - loss: 0.0031 - val_loss: 0.0035
Epoch 11/50
48000/48000 [==============================] - 187s 4ms/step - loss: 0.0030 - val_loss: 0.0028
Epoch 12/50
48000/48000 [==============================] - 188s 4ms/step - loss: 0.0028 - val_loss: 0.0027
Epoch 13/50
48000/48000 [==============================] - 188s 4ms/step - loss: 0.0027 - val_loss: 0.0025
Epoch 14/50
48000/48000 [==============================] - 188s 4ms/step - loss: 0.0026 - val_loss: 0.0025
Epoch 15/50
48000/48000 [==============================] - 188s 4ms/step - loss: 0.0025 - val_loss: 0.0024
Epoch 16/50
48000/48000 [==============================] - 187s 4ms/step - loss: 0.0024 - val_loss: 0.0026
Epoch 17/50
48000/48000 [==============================] - 187s 4ms/step - loss: 0.0023 - val_loss: 0.0023
Epoch 18/50
48000/48000 [==============================] - 186s 4ms/step - loss: 0.0023 - val_loss: 0.0021
Epoch 19/50
48000/48000 [==============================] - 188s 4ms/step - loss: 0.0022 - val_loss: 0.0023
Epoch 20/50
48000/48000 [==============================] - 214s 4ms/step - loss: 0.0022 - val_loss: 0.0023
Epoch 21/50
48000/48000 [==============================] - 193s 4ms/step - loss: 0.0021 - val_loss: 0.0021
Epoch 22/50
48000/48000 [==============================] - 194s 4ms/step - loss: 0.0021 - val_loss: 0.0020
Epoch 23/50
48000/48000 [==============================] - 200s 4ms/step - loss: 0.0020 - val_loss: 0.0019
Epoch 24/50
48000/48000 [==============================] - 203s 4ms/step - loss: 0.0020 - val_loss: 0.0024
Epoch 25/50
48000/48000 [==============================] - 197s 4ms/step - loss: 0.0020 - val_loss: 0.0018
Epoch 26/50
48000/48000 [==============================] - 193s 4ms/step - loss: 0.0019 - val_loss: 0.0019
Epoch 27/50
48000/48000 [==============================] - 192s 4ms/step - loss: 0.0019 - val_loss: 0.0019
Epoch 28/50
48000/48000 [==============================] - 196s 4ms/step - loss: 0.0019 - val_loss: 0.0022
Epoch 29/50
48000/48000 [==============================] - 195s 4ms/step - loss: 0.0019 - val_loss: 0.0018
Epoch 30/50
48000/48000 [==============================] - 194s 4ms/step - loss: 0.0018 - val_loss: 0.0019
Epoch 31/50
48000/48000 [==============================] - 194s 4ms/step - loss: 0.0018 - val_loss: 0.0017
Epoch 32/50
48000/48000 [==============================] - 196s 4ms/step - loss: 0.0018 - val_loss: 0.0018
Epoch 33/50
48000/48000 [==============================] - 196s 4ms/step - loss: 0.0018 - val_loss: 0.0018
Epoch 34/50
48000/48000 [==============================] - 203s 4ms/step - loss: 0.0017 - val_loss: 0.0022
Epoch 35/50
48000/48000 [==============================] - 205s 4ms/step - loss: 0.0017 - val_loss: 0.0017
Epoch 36/50
48000/48000 [==============================] - 209s 4ms/step - loss: 0.0017 - val_loss: 0.0019
Epoch 37/50
48000/48000 [==============================] - 203s 4ms/step - loss: 0.0017 - val_loss: 0.0016
Epoch 38/50
48000/48000 [==============================] - 208s 4ms/step - loss: 0.0017 - val_loss: 0.0018
Epoch 39/50
48000/48000 [==============================] - 209s 4ms/step - loss: 0.0017 - val_loss: 0.0019
Epoch 40/50
48000/48000 [==============================] - 206s 4ms/step - loss: 0.0016 - val_loss: 0.0016
Epoch 41/50
48000/48000 [==============================] - 196s 4ms/step - loss: 0.0016 - val_loss: 0.0015
Epoch 42/50
48000/48000 [==============================] - 197s 4ms/step - loss: 0.0016 - val_loss: 0.0019
Epoch 43/50
48000/48000 [==============================] - 192s 4ms/step - loss: 0.0016 - val_loss: 0.0017
```

```
Epoch 44/50
48000/48000 [==============================] - 197s 4ms/step - loss: 0.0016 - val_loss: 0.0016
Epoch 45/50
48000/48000 [==============================] - 211s 4ms/step - loss: 0.0016 - val_loss: 0.0019
Epoch 46/50
48000/48000 [==============================] - 239s 5ms/step - loss: 0.0016 - val_loss: 0.0017
Epoch 47/50
48000/48000 [==============================] - 288s 6ms/step - loss: 0.0016 - val_loss: 0.0016
Epoch 48/50
48000/48000 [==============================] - 292s 6ms/step - loss: 0.0016 - val_loss: 0.0015
Epoch 49/50
48000/48000 [==============================] - 313s 7ms/step - loss: 0.0015 - val_loss: 0.0016
Epoch 50/50
48000/48000 [==============================] - 297s 6ms/step - loss: 0.0015 - val_loss: 0.0014
```

In [32]:
```
'''
Trained the model on Not-MNIST for 50 epochs,
Plot the loss plot between training and validation data to visualise the model performance
'''
loss = autoencoder_train.history['loss']
val_loss = autoencoder_train.history['val_loss']
epochs = range(epochs)
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```



In [ ]:
```
'''
The validation loss and the training loss both are in sync.
It shows that your model is not overfitting:
the validation loss is decreasing and not increasing, and
there rarely any gap between training and validation loss.

'''
```

In [33]:
```
'''
to reconstruct the test images using the predict() function of Keras and see
how well the model is able reconstruct on the test data

Predicting the trained model on the complete 10,000 test images and plot few of the
reconstructed images to visualize how well
model is able to reconstruct the test images
'''
```

Out[33]:
```
'\nto reconstruct the test images using the predict() function of Keras and see \nhow well the model is able recon
struct on the test data\n\nPredicting the trained model on the complete 10,000 test images and plot few of the \nr
econstructed images to visualize how well\nmodel is able to reconstruct the test images\n'
```

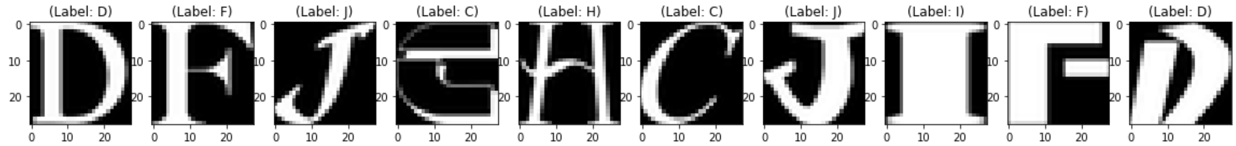
In [34]:
```
pred = autoencoder.predict(test_data)
```

In [35]:
```
pred.shape
```

Out[35]:
```
(10000, 28, 28, 1)
```

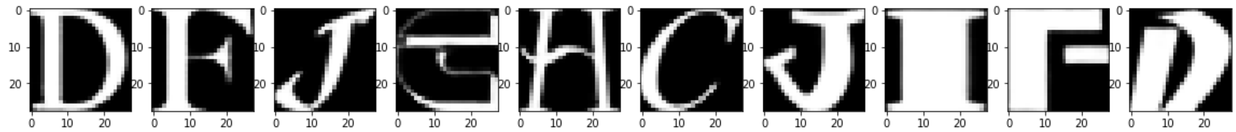
In [36]:
```python
plt.figure(figsize=(20, 4))
print("Test Images")
for i in range(10):
    plt.subplot(2, 10, i+1)
    plt.imshow(test_data[i, ..., 0], cmap='gray')
    curr_lbl = test_labels[i]
    plt.title("(Label: " + str(label_dict[curr_lbl]) + ")")
plt.show()
```

```
Test Images
```



In [37]:
```python
plt.figure(figsize=(20, 4))
print("Reconstruction of Test Images")
for i in range(10):
    plt.subplot(2, 10, i+1)
    plt.imshow(pred[i, ..., 0], cmap='gray')
plt.show()
```

```
Reconstruction of Test Images
```



In [ ]:
```python
'''
model did a fantastic job in reconstructing the test images that you predicted using the model

'''
```