Explore the Autoencoder Architecture:

This is an image classification problem that I want to solve it using an autoencoder.

First, I operated a simple autoencoder.

Import required libraries and Load data from local drive and load dataset

```
# import all the dependencies
from keras.layers import Dense,Conv2D,MaxPooling2D,UpSampling2D
from keras import Input, Model
from keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
```

Build the model:

Then build the model and provide the number of dimensions that will decide how much the input will be compressed. The lesser the dimension, the more will be the compression. The Input is passed through a layer of encoders which are actually a fully connected neural network that also makes the code decoder and hence use the same code for encoding and decoding like an ANN.

```
encoding_dim = 15
input_img = Input(shape=(784,))
# encoded representation of input
encoded = Dense(encoding_dim, activation='relu')(input_img)
# decoded representation of code
decoded = Dense(784, activation='sigmoid')(encoded)
# Model which take input image and shows decoded images
autoencoder = Model(input_img, decoded)
```

Then build the encoder model and decoder model separately so that can easily differentiate between the input and output.

```
# This model shows encoded images
encoder = Model(input_img, encoded)
# Creating a decoder model
encoded_input = Input(shape=(encoding_dim,))
# last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# decoder model
decoder = Model(encoded_input, decoder_layer(encoded_input))
```

Compile the model:

Then we need to compile the model with the ADAM optimizer and cross-entropy loss function fitment.

```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

Load the data:

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
(60000, 784)
(10000, 784)
```

So, we see, we have 784 features, 60000 images, and 10000 classes.

Train the model:

Then I need to train the model on the training data in 15 epochs:

```
autoencoder.fit(x_train, x_train,
                epochs=15,
                batch_size=256,
                validation_data=(x_test, x_test))
```

```
Epoch 1/15
235/235 [==============================] - 2s 8ms/step - loss: 0.3122 - val_loss: 0.2206
Epoch 2/15
235/235 [==============================] - 2s 7ms/step - loss: 0.2002 - val_loss: 0.1849
Epoch 3/15
235/235 [==============================] - 2s 7ms/step - loss: 0.1765 - val_loss: 0.1660
Epoch 4/15
235/235 [==============================] - 2s 7ms/step - loss: 0.1617 - val_loss: 0.1555
Epoch 5/15
235/235 [==============================] - 2s 7ms/step - loss: 0.1535 - val_loss: 0.1489
Epoch 6/15
235/235 [==============================] - 2s 7ms/step - loss: 0.1481 - val_loss: 0.1449
Epoch 7/15
235/235 [==============================] - 2s 7ms/step - loss: 0.1448 - val_loss: 0.1426
Epoch 8/15
235/235 [==============================] - 2s 7ms/step - loss: 0.1426 - val_loss: 0.1405
```

```
235/235 [==============================] - 2s 7ms/step - loss: 0.1426 - val_loss: 0.1405
Epoch 9/15
235/235 [==============================] - 2s 7ms/step - loss: 0.1408 - val_loss: 0.1389
Epoch 10/15
235/235 [==============================] - 2s 7ms/step - loss: 0.1392 - val_loss: 0.1374
Epoch 11/15
235/235 [==============================] - 2s 7ms/step - loss: 0.1379 - val_loss: 0.1361
Epoch 12/15
235/235 [==============================] - 2s 7ms/step - loss: 0.1368 - val_loss: 0.1351
Epoch 13/15
235/235 [==============================] - 2s 7ms/step - loss: 0.1359 - val_loss: 0.1342
Epoch 14/15
235/235 [==============================] - 2s 7ms/step - loss: 0.1352 - val_loss: 0.1335
Epoch 15/15
235/235 [==============================] - 2s 7ms/step - loss: 0.1346 - val_loss: 0.1330
<keras.callbacks.History at 0x7f0b4da25850>
```

As we see, the loss value is about 0.13 and the validation loss is a little smaller than it.

Provide the input and plot the results:

After training, I need to provide the input and I plot the results using the following code:

```
[8]  encoded_img = encoder.predict(x_test)
     decoded_img = decoder.predict(encoded_img)
     plt.figure(figsize=(20, 4))
     for i in range(5):
         # Display original
         ax = plt.subplot(2, 5, i + 1)
         plt.imshow(x_test[i].reshape(28, 28))
         plt.gray()
         ax.get_xaxis().set_visible(False)
         ax.get_yaxis().set_visible(False)
         # Display reconstruction
         ax = plt.subplot(2, 5, i + 1 + 5)
         plt.imshow(decoded_img[i].reshape(28, 28))
         plt.gray()
         ax.get_xaxis().set_visible(False)
         ax.get_yaxis().set_visible(False)
     plt.show()
```

We can clearly see the output for the encoded and decoded images respectively.

Deep CNN Autoencoder:

Since the input here is images, it does make more sense to use a Convolutional Neural network or CNN. The encoder will be made up of a stack of Conv2D and max-pooling layer and the decoder will have a stack of Conv2D and Upsampling Layer.

```
from keras.models import Sequential
model = Sequential()
# encoder network
model.add(Conv2D(30, 3, activation= 'relu', padding='same', input_shape = (28,28,1)))
model.add(MaxPooling2D(2, padding= 'same'))
model.add(Conv2D(15, 3, activation= 'relu', padding='same'))
model.add(MaxPooling2D(2, padding= 'same'))
#decoder network
model.add(Conv2D(15, 3, activation= 'relu', padding='same'))
model.add(UpSampling2D(2))
model.add(Conv2D(30, 3, activation= 'relu', padding='same'))
model.add(UpSampling2D(2))
model.add(Conv2D(1,3,activation='sigmoid', padding= 'same')) # output layer
model.compile(optimizer= 'adam', loss = 'binary_crossentropy')
model.summary()
```

```
max_pooling2d (MaxPooling2D  (None, 14, 14, 30)       0
)

conv2d_1 (Conv2D)            (None, 14, 14, 15)        4065

max_pooling2d_1 (MaxPooling  (None, 7, 7, 15)         0
2D)

conv2d_2 (Conv2D)            (None, 7, 7, 15)          2040

up_sampling2d (UpSampling2D  (None, 14, 14, 15)       0
)

conv2d_3 (Conv2D)            (None, 14, 14, 30)        4080

up_sampling2d_1 (UpSampling  (None, 28, 28, 30)       0
2D)

conv2d_4 (Conv2D)            (None, 28, 28, 1)         271

=================================================================
Total params: 10,756
Trainable params: 10,756
Non-trainable params: 0
```

So, this model has 10,760 parameters and all of them are trainable and have no non-trainable parameters.

Load the data and train the model:

Now we need to load the data and train the model

```
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))
model.fit(x_train, x_train,
          epochs=15,
          batch_size=128,
          validation_data=(x_test, x_test))
```

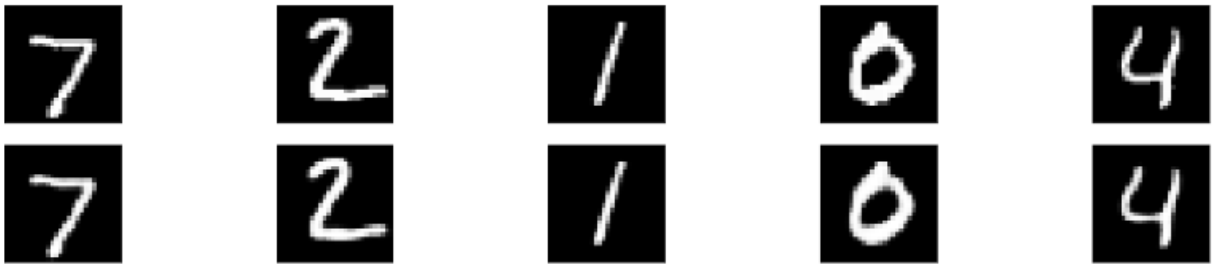
```
469/469 [==============================] - 86s 183ms/step - loss: 0.0733 - val_loss: 0.0719
Epoch 5/15
469/469 [==============================] - 87s 185ms/step - loss: 0.0721 - val_loss: 0.0709
Epoch 6/15
469/469 [==============================] - 87s 186ms/step - loss: 0.0711 - val_loss: 0.0701
Epoch 7/15
469/469 [==============================] - 87s 185ms/step - loss: 0.0704 - val_loss: 0.0695
Epoch 8/15
469/469 [==============================] - 87s 185ms/step - loss: 0.0699 - val_loss: 0.0691
Epoch 9/15
469/469 [==============================] - 87s 185ms/step - loss: 0.0695 - val_loss: 0.0687
Epoch 10/15
469/469 [==============================] - 87s 185ms/step - loss: 0.0691 - val_loss: 0.0684
Epoch 11/15
469/469 [==============================] - 88s 188ms/step - loss: 0.0688 - val_loss: 0.0683
Epoch 12/15
469/469 [==============================] - 88s 187ms/step - loss: 0.0685 - val_loss: 0.0679
Epoch 13/15
469/469 [==============================] - 87s 185ms/step - loss: 0.0683 - val_loss: 0.0677
Epoch 14/15
469/469 [==============================] - 87s 185ms/step - loss: 0.0681 - val_loss: 0.0675
Epoch 15/15
469/469 [==============================] - 87s 186ms/step - loss: 0.0679 - val_loss: 0.0673
<keras.callbacks.History at 0x7f0b48407310>
```

As we see, the loss value is about 0.067 and the validation loss is a little smaller than it.

The training and validation loss for deep CNN are lower than these values for simple autoencoder.

Now provide the input and plot the output for the following results:

```
pred = model.predict(x_test)
plt.figure(figsize=(20, 4))
for i in range(5):
    # Display original
    ax = plt.subplot(2, 5, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # Display reconstruction
    ax = plt.subplot(2, 5, i + 1 + 5)
    plt.imshow(pred[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



Comparing the above result with the output of a simple autoencoder, we see that the error for deep CNN is lower, and the images are clearer.

Denoising Autoencoder:

Now we will see how the model performs with noise in the image. What we mean by noise is blurry images, changing the color of the images, or even white markers on the image.

```
noise_factor = 0.7
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
#Here is how the noisy images look right now.
plt.figure(figsize=(20, 2))
for i in range(1, 5 + 1):
    ax = plt.subplot(1, 5, i)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Now the images are barely identifiable and to increase the extent of the autoencoder, we will modify the layers of the defined model to increase the filter so that the model performs better and then fit the model.

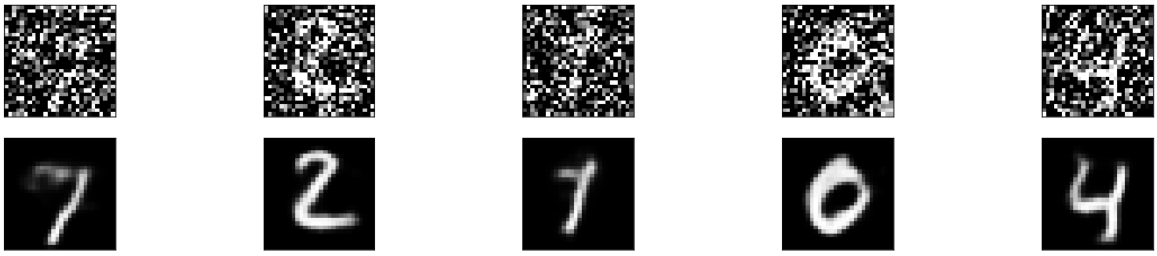
```
model = Sequential()
# encoder network
model.add(Conv2D(35, 3, activation= 'relu', padding='same', input_shape = (28,28,1)))
model.add(MaxPooling2D(2, padding= 'same'))
model.add(Conv2D(25, 3, activation= 'relu', padding='same'))
model.add(MaxPooling2D(2, padding= 'same'))
#decoder network
model.add(Conv2D(25, 3, activation= 'relu', padding='same'))
model.add(UpSampling2D(2))
model.add(Conv2D(35, 3, activation= 'relu', padding='same'))
model.add(UpSampling2D(2))
model.add(Conv2D(1,3,activation='sigmoid', padding= 'same')) # output layer
model.compile(optimizer= 'adam', loss = 'binary_crossentropy')
model.fit(x_train_noisy, x_train,
          epochs=15,
          batch_size=128,
          validation_data=(x_test_noisy, x_test))
```

```
Epoch 6/15
469/469 [==============================] - 107s 228ms/step - loss: 0.1291 - val_loss: 0.1272
Epoch 7/15
469/469 [==============================] - 107s 228ms/step - loss: 0.1278 - val_loss: 0.1265
Epoch 8/15
469/469 [==============================] - 107s 227ms/step - loss: 0.1266 - val_loss: 0.1254
Epoch 9/15
469/469 [==============================] - 107s 228ms/step - loss: 0.1258 - val_loss: 0.1250
Epoch 10/15
469/469 [==============================] - 107s 228ms/step - loss: 0.1250 - val_loss: 0.1239
Epoch 11/15
469/469 [==============================] - 107s 228ms/step - loss: 0.1243 - val_loss: 0.1235
Epoch 12/15
469/469 [==============================] - 107s 228ms/step - loss: 0.1238 - val_loss: 0.1229
Epoch 13/15
469/469 [==============================] - 109s 232ms/step - loss: 0.1233 - val_loss: 0.1223
Epoch 14/15
469/469 [==============================] - 107s 227ms/step - loss: 0.1229 - val_loss: 0.1222
Epoch 15/15
469/469 [==============================] - 107s 227ms/step - loss: 0.1225 - val_loss: 0.1223
<keras.callbacks.History at 0x7f0b455d4890>
```

As we see, the loss value is about 0.122 and the validation loss is a little smaller.

After the training, we will provide the input and write a plot function to see the final results.

```
pred = model.predict(x_test_noisy)
plt.figure(figsize=(20, 4))
for i in range(5):
    # Display original
    ax = plt.subplot(2, 5, i + 1)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # Display reconstruction
    ax = plt.subplot(2, 5, i + 1 + 5)
    plt.imshow(pred[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



We have gone through the structure of how autoencoders work and worked with 3 types of autoencoders. There are multiple uses for autoencoders like dimensionality reduction image compression, recommendations system for movies and songs, and more. The performance of the model can be increased by training it for more epochs or also by increasing the dimension of our network.

Reference:

https://www.analyticsvidhya.com/blog/2021/06/complete-guide-on-how-to-use-autoencoders-in-python/