Gurjus Singh

November 15th, 2020

MSDS 422 – Practical Machine Learning

Assignment #9 Auto Encoder

Data preparation, exploration, visualization

For the Data Analysis this week, I again dealt with the MNIST dataset from Week 5 and Week 6. The MNIST data is a dataset where each row has 784 pixels/features of numbers 0-9. What my goal for this Data Analysis was to use MNIST data and do **Dimensionality Reduction similar to PCA, but with Autoencoders [1]. This is a key unsupervised learning method, which means it does not use a Target/Label to train [1]. The key difference between PCA and Autoencoders is that PCA deals with Linear Combinations when extracting features while Autoencoders deal with nonlinear combinations [1]. I used Variational Autoencoders which not only extracts features, but also makes a distribution of the images it sees and uses that to create a random sample [2].**

There was not much data prep that needed to be done. The first thing I did was imported the TensorFlow and Keras packages as before. I then checked the versions as Tensorflow 2.0 or above was required for this Analysis. I then defined variables for my layers so my model could define which later computed what statistics per layer or used what activation function. After defining explicitly, the layers I then defined a sampling function which was used to generate more data points, and then I defined the model creating the encoder, decoder and instantiating the VAE model.

After defining the model, I then loaded in the MNIST data splitting it into training and test sets after downloading. Next, I need to convert the data to float 32 and divide by 255 so it was in pixel type. I also looked the shape after each split after words to see the labels and if there were exactly 784 features. Output 1-1 showed that there were 60,000 rows with 784 columns and 60,000 labels as well in the train set. I had split up the data 60,000-10,000 as shown in 1-1.

```
print('x_train:\t{}'.format(x_train.shape))
print('y_train:\t{}'.format(y_train.shape))
print('x_test:\t\t{}'.format(x_test.shape))
print('y_test:\t\t{}'.format(y_test.shape))
```

```
x_train:        (60000, 784)
y_train:        (60000,)
x_test:         (10000, 784)
y_test:         (10000,)
```

Output 1-1

I also saw if the dataset was balanced in 1-2 as I did in the last analysis, to see if the data set was balanced with all the numbers, I could see the training set was pretty balanced with the most being 1s. Same thing with the test data, although the 5's did seem pretty low. This was good to know as well to see how our normal distribution would turn out using the Bivariate Standard Normal [2]. I also looked at the first 10 labels for training set.

```
print("First ten labels training dataset:\n {}\n".format(y_train[0:10]))
```

```
First ten labels training dataset:
 [5 0 4 1 9 2 1 3 1 4]
```

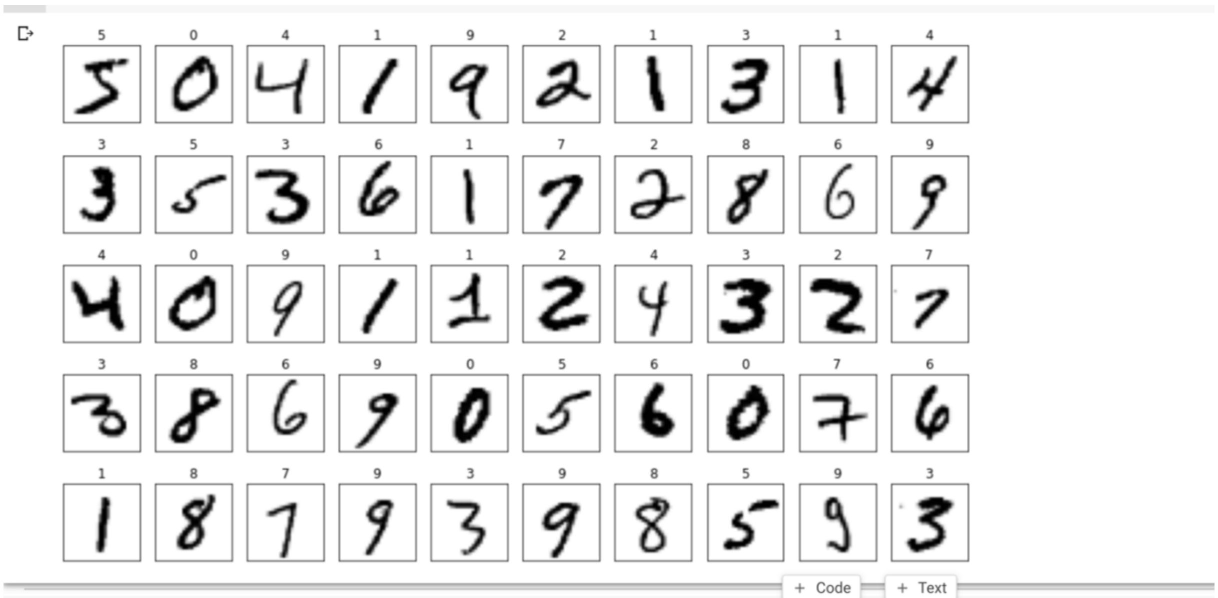
```
Counter(y_train).most_common()
```

```
[(1, 6742),
 (7, 6265),
 (3, 6131),
 (2, 5958),
 (9, 5949),
 (0, 5923),
 (6, 5918),
 (8, 5851),
 (4, 5842),
 (5, 5421)]
```

```
Counter(y_test).most_common()
```

```
[(1, 1135),
 (2, 1032),
 (7, 1028),
 (3, 1010),
 (9, 1009),
 (4, 982),
 (0, 980),
 (8, 974),
 (6, 958),
 (5, 892)]
```

Output 1-2



Output 1-3

In 1-3, I also saw what the images looked like when the pixels were stitched together. I did see some images that clearly looked the same as labels, while others looked different than their labels. After doing this it was time to train the models.

Review research design and modeling methods

For this assignment I focused, on using Variational Auto Encoders with Intermediate Size of 16 neurons and 64 neurons respectively. As mentioned in the data prep section above I split up the dataset into a 60,000-10,000 split. As also mentioned above, Auto Encoders are generally used for dimensionality reduction, but is also a neural network so it is two in one approach unlike PCA where you have to do dimensionality reduction first and then feed into a training/learning algorithm [2]. Another thing that Auto Encoder are especially specialized with is unlike PCA they are nonlinear when extracting features [1]. Autoencoders also utilizes the sigmoid function which is a function that varies between 0 to 1 because I need to figure out a probability for each of random numbers to show where they would be in the Monte Carlo simulation/histogram of the distribution and which number they would correspond to after doing the random sampling [2].

For the loss function of the Variational Auto Encoder there is a reconstruction loss which is used to make sure Auto Encoder "reconstructs the inputs" [3]. Another part of the loss function is the latent loss to make sure the Autoencoder is sampling for a Gaussian Normal Distribution [3]. We also take into account the regularization loss as well [2]. For this analysis these first two losses also needed to be custom made and I judged my models based on the loss because there was no accuracy score as this was unsupervised.

Review Results and Evaluate Model

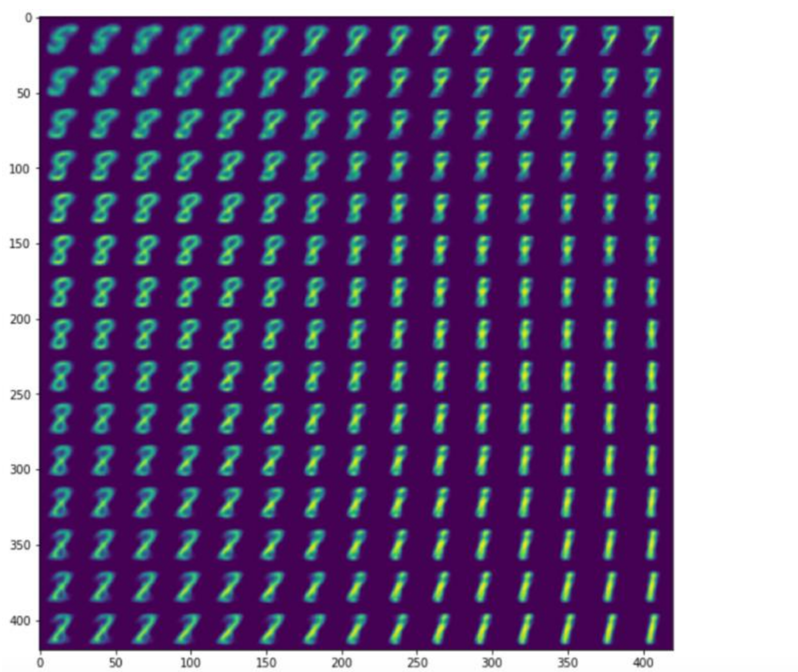
After fitting the two models for the first model I got a loss around 144 as seen in 1-4, while my loss for model 2 was around 155 as seen in 1-5. This showed that Model 1 was performing better with 64 intermediate neurons. I also looked at the latent space which helped to generate the images and how the features were extracted in 1-6, 1-7. After looking at the latent spaces, I tried to see how the distribution was on train set and test set for both models in 1-8 through 1-11 making a bivariate plot.

```
1875/1875 [==============================] - 5s 3ms/step - loss: 145.5450 - val_loss: 147.1726
Epoch 80/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.5238 - val_loss: 147.1579
Epoch 81/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.5121 - val_loss: 147.3082
Epoch 82/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.4267 - val_loss: 147.0922
Epoch 83/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.3851 - val_loss: 147.0584
Epoch 84/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.3466 - val_loss: 147.1474
Epoch 85/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.3132 - val_loss: 146.7194
Epoch 86/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.2850 - val_loss: 147.2583
Epoch 87/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.2377 - val_loss: 146.7732
Epoch 88/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.1782 - val_loss: 147.0087
Epoch 89/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.1775 - val_loss: 147.0368
Epoch 90/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.1227 - val_loss: 147.4142
Epoch 91/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.0991 - val_loss: 146.8947
Epoch 92/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.0788 - val_loss: 146.9851
Epoch 93/100
1875/1875 [==============================] - 5s 3ms/step - loss: 144.9976 - val_loss: 147.2339
Epoch 94/100
1875/1875 [==============================] - 5s 3ms/step - loss: 145.0135 - val_loss: 146.8443
Epoch 95/100
1875/1875 [==============================] - 5s 3ms/step - loss: 144.9243 - val_loss: 147.1355
Epoch 96/100
1875/1875 [==============================] - 5s 3ms/step - loss: 144.9264 - val_loss: 146.6772
Epoch 97/100
1875/1875 [==============================] - 5s 3ms/step - loss: 144.8852 - val_loss: 146.9704
Epoch 98/100
1875/1875 [==============================] - 5s 3ms/step - loss: 144.8646 - val_loss: 146.9803
Epoch 99/100
1875/1875 [==============================] - 5s 3ms/step - loss: 144.7992 - val_loss: 146.8541
Epoch 100/100
1875/1875 [==============================] - 5s 3ms/step - loss: 144.7910 - val_loss: 146.8879
```

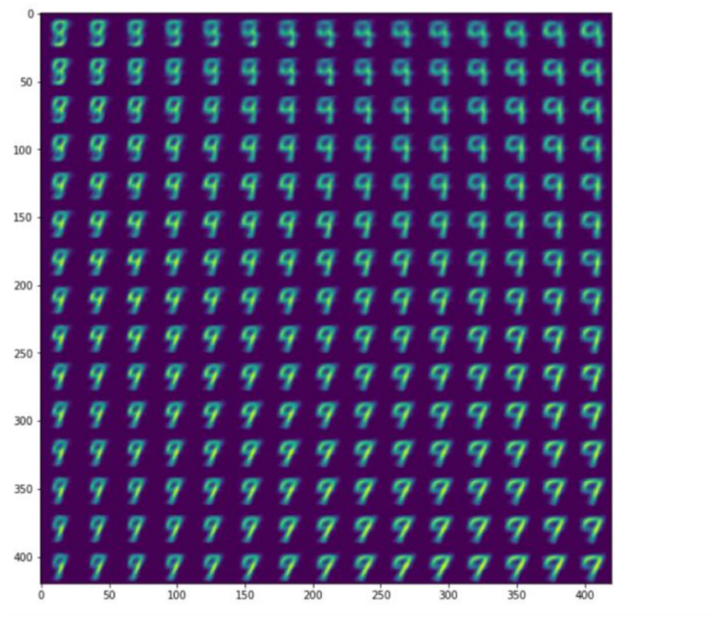
Model 1 64 neurons 1-4

```
1875/1875 [==============================] - 5s 3ms/step - loss: 155.3023 - val_loss: 156.2650
Epoch 86/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.2603 - val_loss: 156.0853
Epoch 87/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.2447 - val_loss: 156.0485
Epoch 88/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.2269 - val_loss: 156.0775
Epoch 89/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.2316 - val_loss: 156.1007
Epoch 90/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.2127 - val_loss: 156.2241
Epoch 91/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.1626 - val_loss: 156.1383
Epoch 92/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.1701 - val_loss: 156.1400
Epoch 93/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.1236 - val_loss: 156.2285
Epoch 94/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.1186 - val_loss: 156.0569
Epoch 95/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.1096 - val_loss: 155.9551
Epoch 96/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.0616 - val_loss: 155.9550
Epoch 97/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.0635 - val_loss: 156.0217
Epoch 98/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.0468 - val_loss: 156.0954
Epoch 99/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.0234 - val_loss: 155.9130
Epoch 100/100
1875/1875 [==============================] - 5s 3ms/step - loss: 155.0128 - val_loss: 155.9842
```

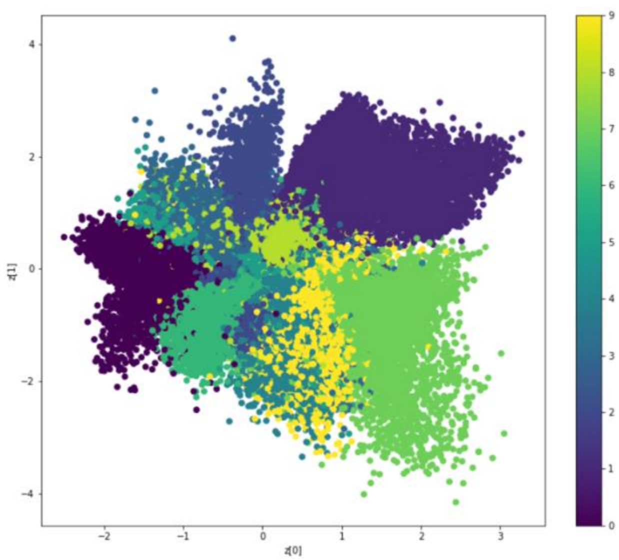
Model 2 16 neurons 1-5



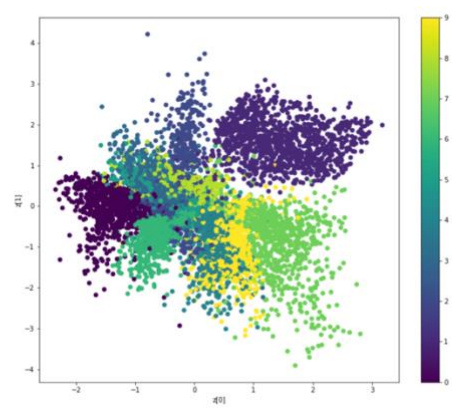
Model 1 Images 1-6



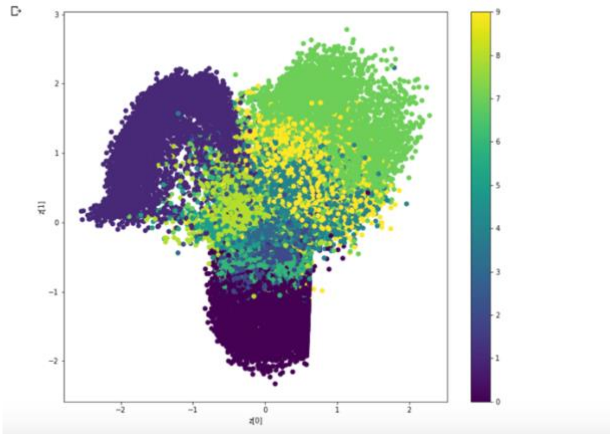
Model 2 Images 1-7

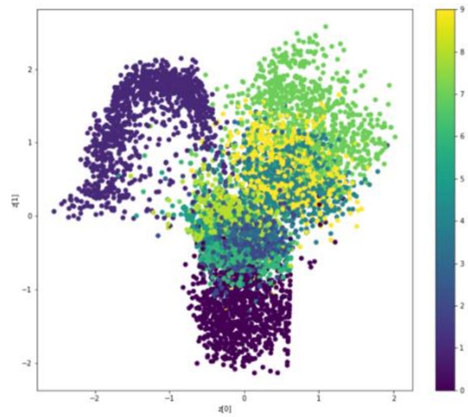


Model 1 Bivar Plot Train Data 1-8



Model 1 Bivar Plot Test Data 1-9





Model 2 Train Data Plot 1-10 *Model 2 Test Data Plot 1-11*

For model 1 it identified the majority of labels in the Gaussian distribution were associated with 1s and 7s for the Train Data as seen in 1-8, while for model 2 it said exactly the same thing for Train Data in 1-10. As you can see for the Test Data in both models it had the same type of shape as the Test Data. The shape is basically is the beast form of compression of the data that was chosen by the autoencoder in the intermediate layer when figuring out the latent space, feature extraction, and dimensional reduction [2].

Implementation and Programming

In this analysis the main packages that were used were the **Tensorflow and Keras packages** for using the Autoencoders and plotting the Bivariate plots mentioned in the last section. I also utilized **numpy and matplotlib**. All the packages imported are seen below in 1-12. I first checked the packages by using attribute **tf.__version__** and I saw that my versions for Tensorflow was 2.3.0 and I used the same type of syntax for Keras which spit out 2.4.0. I then created variables for the dimensions for each layer **28*28** for input layer which signified the original 784 features. The intermediate_dim layer was 16, and 64 for both models. The Latent Dimensions were 2. I then created layers using **keras.Input, and layers.Dense** which would help compute mean and standard dev when instantiating and running the models. I then create a sampling function which would be used in the model when sampling at random to generate numbers for Gaussian Distribution based on MNIST data.

Before running the model I created the encoder, decoder which was used by using the syntax **keras.Model()** and using the inputs. I also created the latent inputs using the 64 neurons mentioned above and using **layers.Dense()** and I also used this for the outputs as well which transformed the extracted features back in 784 features which was used to generate more data.

I then instantiated the model using the same syntax above **keras.Model()** These objects were all in the **Keras and Tensorflow** packages. I then saw a summary of the model using **summary** function and using **keras.utils.plot_model** to save a diagram of the model. These were also from the **Keras and Tensorflow** packages. I then downloaded the MNIST dataset using **Tensorflow's load_data** function. I then created custom loss as the loss was very different from losses I have been using in other analyses. I particularly used Tensorflow functions such as **K.mean(), K.sum(), K.square, and add_loss()** for computing these losses as seen in 1-13 below.

After I then feature engineering by converting the features to float 32, and divided them by 255 and I took the product to reshape then using **np.product** from **numpy package.** Then after that I wanted to see the shape of the data, which using **.shape attribute.** I also saw most initial 10 labels of data by using index slicing **[0:10].** I also wanted to see most common labels using **Counter(),most_common().** Lastly I saw the original images by using **imshow** from **matplotlib package** and saw the images by plotting their pixels on a subplots of which I saw 50 images. After seeing the initial data, I then used **.fit() function**. to train data. I used loss to compare both models. I then used **matplotlib and imshow()** function and used nested **for loops** to plot latent space. I made custom function which used **.predict** function from **Tensorflow** package to make a bivariate plot and used **.scatter from matplotlib package** to plot the data points taken from predict function. The .predict function had three columns which included the latent features which I plotted from predict function.

```
# Helper libraries
import datetime
from packaging import version
import matplotlib.pyplot as plt
import seaborn as sns

from collections import Counter
import numpy as np
import pandas as pd

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.datasets import mnist
from tensorflow.keras import backend as K
```

Import Packages 1-12

```
[ ]
reconstruction_loss = keras.losses.binary_crossentropy(inputs, outputs)
reconstruction_loss *= original_dim
kl_loss = 1 + z_log_sigma - K.square(z_mean) - K.exp(z_log_sigma)
kl_loss = K.sum(kl_loss, axis=-1)
kl_loss *= -0.5
vae_loss = K.mean(reconstruction_loss + kl_loss)
vae.add_loss(vae_loss)
vae.compile(optimizer='adam')
```

Loss computation 1-13

Exposition, problem description, Management recommendation

The goal as mentioned before was to use an unsupervised method in Auto Encoders on the MNIST Dataset. My results suggested by looking at 1-4 and 1-5 that Model 1 with 64 neurons in intermediate layer was performing better with a loss of 144. Model 2 with 16 neurons had a loss of 155. **Therefore, I recommend to management to choose Model 1 in 1-4 with 64 neurons in the Intermediate Layer and represented below in 1-14.** In the future I hope to try to find a way to lower the loss function even more as I felt it was still too high for my liking. Maybe getting it down to 100 or under 100 would be nicer. I do realize that I have to be cautious in taking this approach as if I significantly reduce the loss it would cause overfitting problems [1]. I also was satisfied with the fitting as it was not as time consuming as my other analysis. I also saw some beautiful bivariate charts as seen above which I like and hope to analyze them more in depth in the future.

```
[ ] original_dim = 28 * 28
    intermediate_dim = 64
    latent_dim = 2

    inputs = keras.Input(shape=(original_dim,))
    h = layers.Dense(intermediate_dim, activation='relu')(inputs)
    z_mean = layers.Dense(latent_dim)(h)
    z_log_sigma = layers.Dense(latent_dim)(h)
```

Model 1 – 1-14

References

[1] Srinivasan, S. (2020). *Week 9 Lecture Auto Encoders* [Slides]. Canvas. https://canvas.northwestern.edu/courses/125893/files/9374628/download?wrap=1

[2] Srinivasan, S. (2020b, November 10). *Sync Session Auto Encoder* [Slides]. Canvas. https://canvas.northwestern.edu/courses/125893/files/9968055?module_item_id=1703375

[3] Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems* (2nd ed.). O'Reilly Media.

APPENDIX BELOW

```
# Helper libraries
import datetime
from packaging import version
import matplotlib.pyplot as plt
import seaborn as sns

from collections import Counter
import numpy as np
import pandas as pd

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.datasets import mnist
from tensorflow.keras import backend as K
```

```
%matplotlib inline
np.set_printoptions(precision=3, suppress=True)
```

```
print("This notebook requires TensorFlow 2.0 or above")
print("TensorFlow version: ", tf.__version__)
assert version.parse(tf.__version__).release[0] >=2
```

```
This notebook requires TensorFlow 2.0 or above
TensorFlow version:  2.3.0
```

```
print("Keras version: ", keras.__version__)
```

```
Keras version:  2.4.0
```

```
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
original_dim = 28 * 28
intermediate_dim = 64
latent_dim = 2

inputs = keras.Input(shape=(original_dim,))
h = layers.Dense(intermediate_dim, activation='relu')(inputs)
z_mean = layers.Dense(latent_dim)(h)
z_log_sigma = layers.Dense(latent_dim)(h)
```

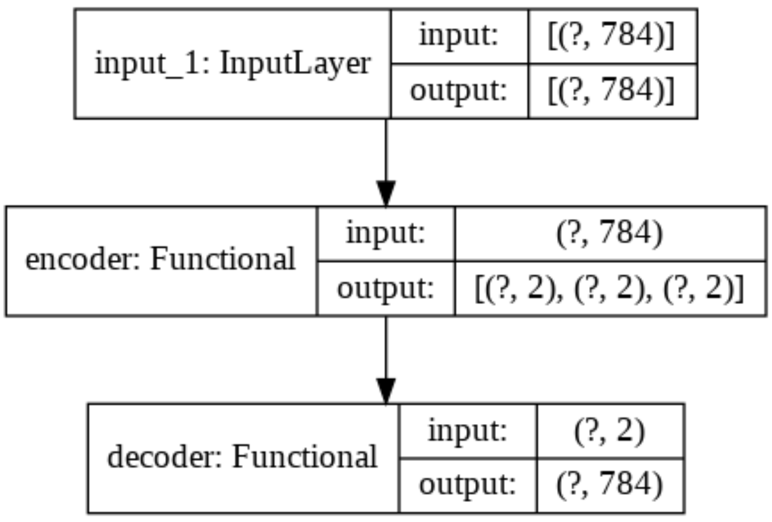
```
def sampling(args):
    z_mean, z_log_sigma = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
                              mean=0., stddev=0.1)
    return z_mean + K.exp(z_log_sigma) * epsilon

z = layers.Lambda(sampling)([z_mean, z_log_sigma])
```

```
# Create encoder
encoder = keras.Model(inputs, [z_mean, z_log_sigma, z], name='encoder')

# Create decoder
latent_inputs = keras.Input(shape=(latent_dim,), name='z_sampling')
x = layers.Dense(intermediate_dim, activation='relu')(latent_inputs)
outputs = layers.Dense(original_dim, activation='sigmoid')(x)
decoder = keras.Model(latent_inputs, outputs, name='decoder')

# instantiate VAE model
outputs = decoder(encoder(inputs)[2])
vae = keras.Model(inputs, outputs, name='vae_mlp')
```

```
vae.summary()
```

```
Model: "vae_mlp"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 784)]             0
_________________________________________________________________
encoder (Functional)         [(None, 2), (None, 2), (N 50500
_________________________________________________________________
decoder (Functional)         (None, 784)               51152
=================================================================
Total params: 101,652
Trainable params: 101,652
Non-trainable params: 0
_________________________________________________________________
```

```
keras.utils.plot_model(vae, "EncoderModel.png", show_shapes=True)
```




```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
```

```
reconstruction_loss = keras.losses.binary_crossentropy(inputs, outputs)
reconstruction_loss *= original_dim
kl_loss = 1 + z_log_sigma - K.square(z_mean) - K.exp(z_log_sigma)
kl_loss = K.sum(kl_loss, axis=-1)
kl_loss *= -0.5
vae_loss = K.mean(reconstruction_loss + kl_loss)
vae.add_loss(vae_loss)
vae.compile(optimizer='adam')
```

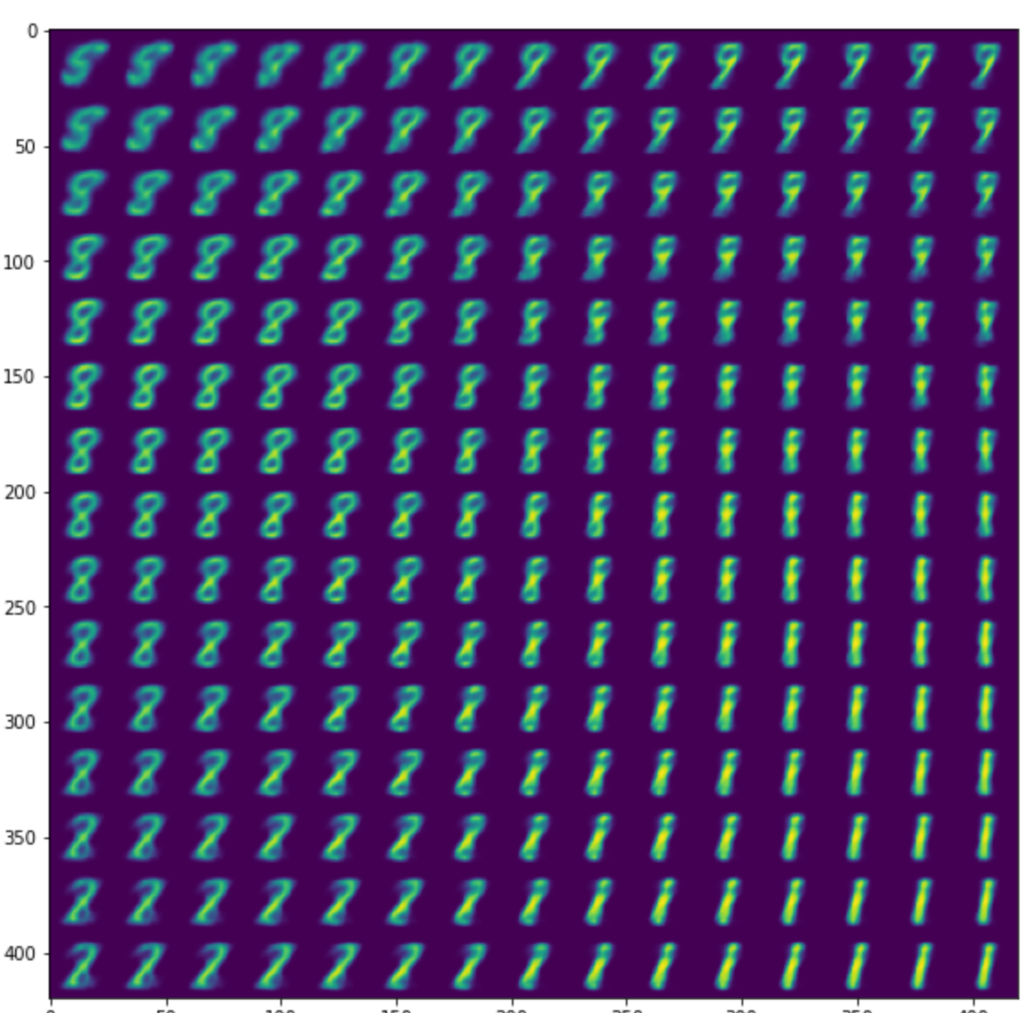
```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
```

```
history = vae.fit(x_train, x_train,
            epochs=100,
            batch_size=32,
            validation_data=(x_test, x_test))
```

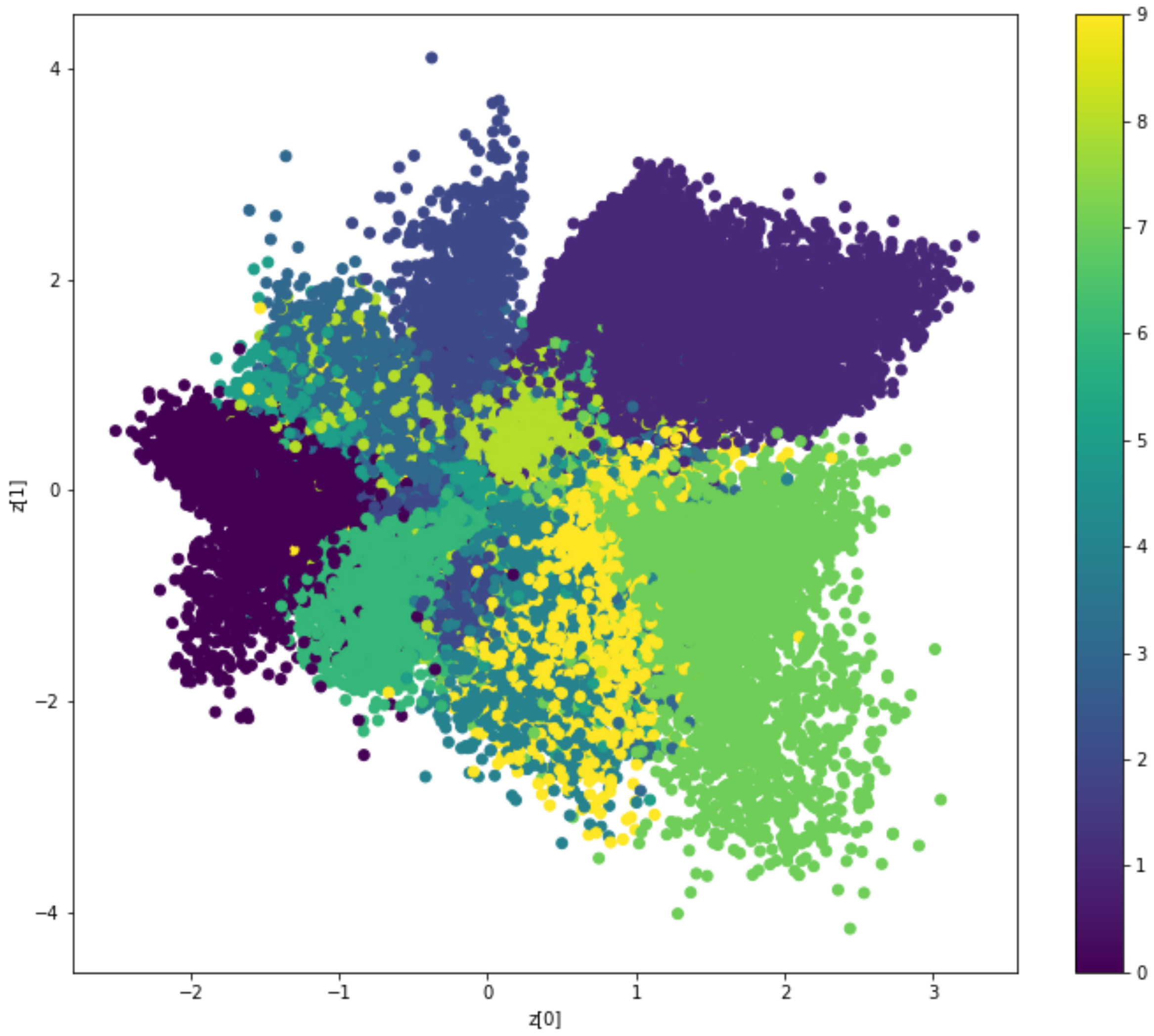
```
1875/1875 [==============================] - 5s 3ms/step - loss: 148.3130 - val_loss: 148.6449
Epoch 40/100
1875/1875 [==============================] - 5s 3ms/step - loss: 148.2072 - val_loss: 148.9730
Epoch 41/100
1875/1875 [==============================] - 5s 3ms/step - loss: 148.0763 - val_loss: 148.7733
Epoch 42/100
1875/1875 [==============================] - 5s 3ms/step - loss: 147.9726 - val_loss: 148.7710
Epoch 43/100
1875/1875 [==============================] - 5s 3ms/step - loss: 147.8771 - val_loss: 148.4862
Epoch 44/100
1875/1875 [==============================] - 5s 3ms/step - loss: 147.7949 - val_loss: 148.4550
Epoch 45/100
1875/1875 [==============================] - 5s 3ms/step - loss: 147.6966 - val_loss: 148.4832
Epoch 46/100
1875/1875 [==============================] - 5s 3ms/step - loss: 147.5869 - val_loss: 148.2121
Epoch 47/100
1875/1875 [==============================] - 5s 3ms/step - loss: 147.4786 - val_loss: 148.5752
Epoch 48/100
1875/1875 [==============================] - 5s 3ms/step - loss: 147.4017 - val_loss: 148.4205
Epoch 49/100
1875/1875 [==============================] - 5s 3ms/step - loss: 147.3118 - val_loss: 148.2887
Epoch 50/100
1875/1875 [==============================] - 5s 3ms/step - loss: 147.2359 - val_loss: 148.3508
Epoch 51/100
1875/1875 [==============================] - 5s 3ms/step - loss: 147.1363 - val_loss: 148.0448
Epoch 52/100
1875/1875 [==============================] - 5s 3ms/step - loss: 147.0728 - val_loss: 148.1633
Epoch 53/100
1875/1875 [==============================] - 5s 3ms/step - loss: 147.0213 - val_loss: 148.0405
Epoch 54/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.9193 - val_loss: 148.0766
Epoch 55/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.8703 - val_loss: 148.0860
Epoch 56/100
```

```
1875/1875 [==============================] - 5s 3ms/step - loss: 146.7978 - val_loss: 147.9145
Epoch 57/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.7256 - val_loss: 147.8531
Epoch 58/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.6671 - val_loss: 148.3363
Epoch 59/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.6176 - val_loss: 147.9139
Epoch 60/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.5292 - val_loss: 147.8346
Epoch 61/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.4833 - val_loss: 147.5485
Epoch 62/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.3974 - val_loss: 147.9326
Epoch 63/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.3536 - val_loss: 147.7632
Epoch 64/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.2903 - val_loss: 147.7570
Epoch 65/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.2692 - val_loss: 148.3321
Epoch 66/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.2062 - val_loss: 147.5096
Epoch 67/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.1156 - val_loss: 147.6299
Epoch 68/100
1875/1875 [==============================] - 5s 3ms/step - loss: 146.0764 - val_loss: 147.8277
Epoch 69/100
```

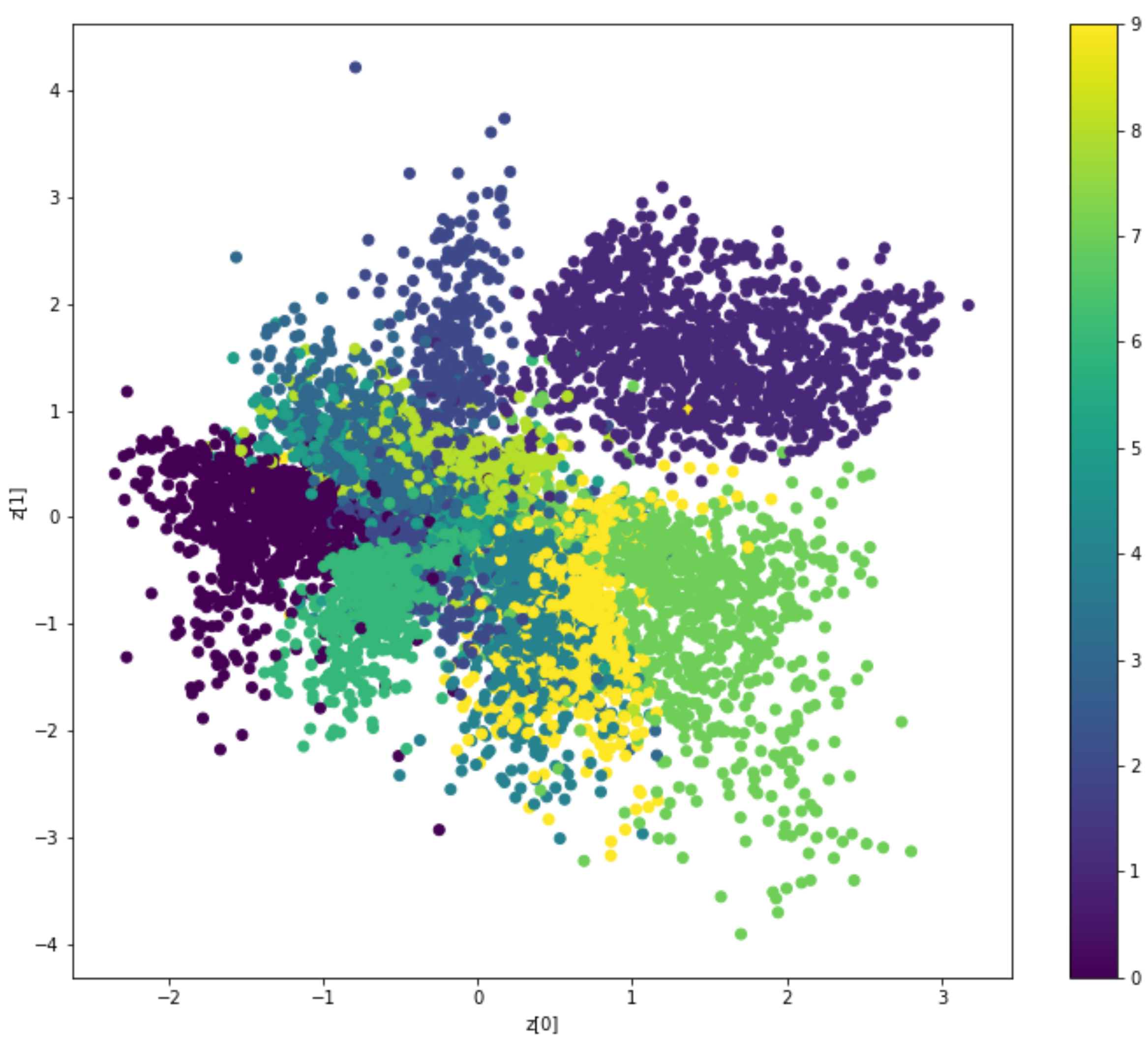
```python
# Display a 2D manifold of the digits
n = 15  # figure with 15x15 digits
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
grid_x = np.linspace(0.05, 0.95, n)
grid_y = np.linspace(0.05, 0.95, n)

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]])
        x_decoded = decoder.predict(z_sample)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure)
plt.show()
```




```python
def plot_label_clusters(encoder, decoder, data, labels):
    # display a 2D plot of the digit classes in the latent space
    z_mean, _, _ = encoder.predict(data)
    plt.figure(figsize=(12, 10))
    plt.scatter(z_mean[:, 0], z_mean[:, 1], c=labels)
    plt.colorbar()
    plt.xlabel("z[0]")
    plt.ylabel("z[1]")
    plt.show()

plot_label_clusters(encoder, decoder, x_train, y_train)
```




```
plot_label_clusters(encoder, decoder, x_test, y_test)
```




```
original_dim = 28 * 28
intermediate_dim = 16
latent_dim = 2

inputs = keras.Input(shape=(original_dim,))
h = layers.Dense(intermediate_dim, activation='relu')(inputs)
z_mean = layers.Dense(latent_dim)(h)
z_log_sigma = layers.Dense(latent_dim)(h)
```

```
def sampling(args):
    z_mean, z_log_sigma = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
                              mean=0., stddev=0.1)
    return z_mean + K.exp(z_log_sigma) * epsilon

z = layers.Lambda(sampling)([z_mean, z_log_sigma])
```

```
# Create encoder
encoder = keras.Model(inputs, [z_mean, z_log_sigma, z], name='encoder')
```

```
# Create decoder
latent_inputs = keras.Input(shape=(latent_dim,), name='z_sampling')
x = layers.Dense(intermediate_dim, activation='relu')(latent_inputs)
outputs = layers.Dense(original_dim, activation='sigmoid')(x)
decoder = keras.Model(latent_inputs, outputs, name='decoder')

# instantiate VAE model
outputs = decoder(encoder(inputs)[2])
vae = keras.Model(inputs, outputs, name='vae_mlp')
```

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

```
print('x_train:\t{}'.format(x_train.shape))
print('y_train:\t{}'.format(y_train.shape))
print('x_test:\t\t{}'.format(x_test.shape))
print('y_test:\t\t{}'.format(y_test.shape))
```

```
x_train:        (60000, 784)
y_train:        (60000,)
x_test:         (10000, 784)
y_test:         (10000,)
```

```
print("First ten labels training dataset:\n {}\n".format(y_train[0:10]))
```

```
First ten labels training dataset:
 [5 0 4 1 9 2 1 3 1 4]
```

```
Counter(y_train).most_common()
```

```
[(1, 6742),
 (7, 6265),
 (3, 6131),
 (2, 5958),
 (9, 5949),
 (0, 5923),
 (6, 5918),
 (8, 5851),
 (4, 5842),
 (5, 5421)]
```

```
Counter(y_test).most_common()
```

```
[(1, 1135),
 (2, 1032),
 (7, 1028),
 (3, 1010),
 (9, 1009),
 (4, 982),
 (0, 980),
 (8, 974),
 (6, 958),
 (5, 892)]
```

```
fig = plt.figure(figsize = (15, 9))

for i in range(50):
    plt.subplot(5, 10, 1+i)
    plt.title(y_train[i])
    plt.xticks([])
    plt.yticks([])
    plt.imshow(x_train[i].reshape(28,28), cmap='binary')
```


```
reconstruction_loss = keras.losses.binary_crossentropy(inputs, outputs)
reconstruction_loss *= original_dim
kl_loss = 1 + z_log_sigma - K.square(z_mean) - K.exp(z_log_sigma)
kl_loss = K.sum(kl_loss, axis=-1)
kl_loss *= -0.5
vae_loss = K.mean(reconstruction_loss + kl_loss)
vae.add_loss(vae_loss)
vae.compile(optimizer='adam')
```

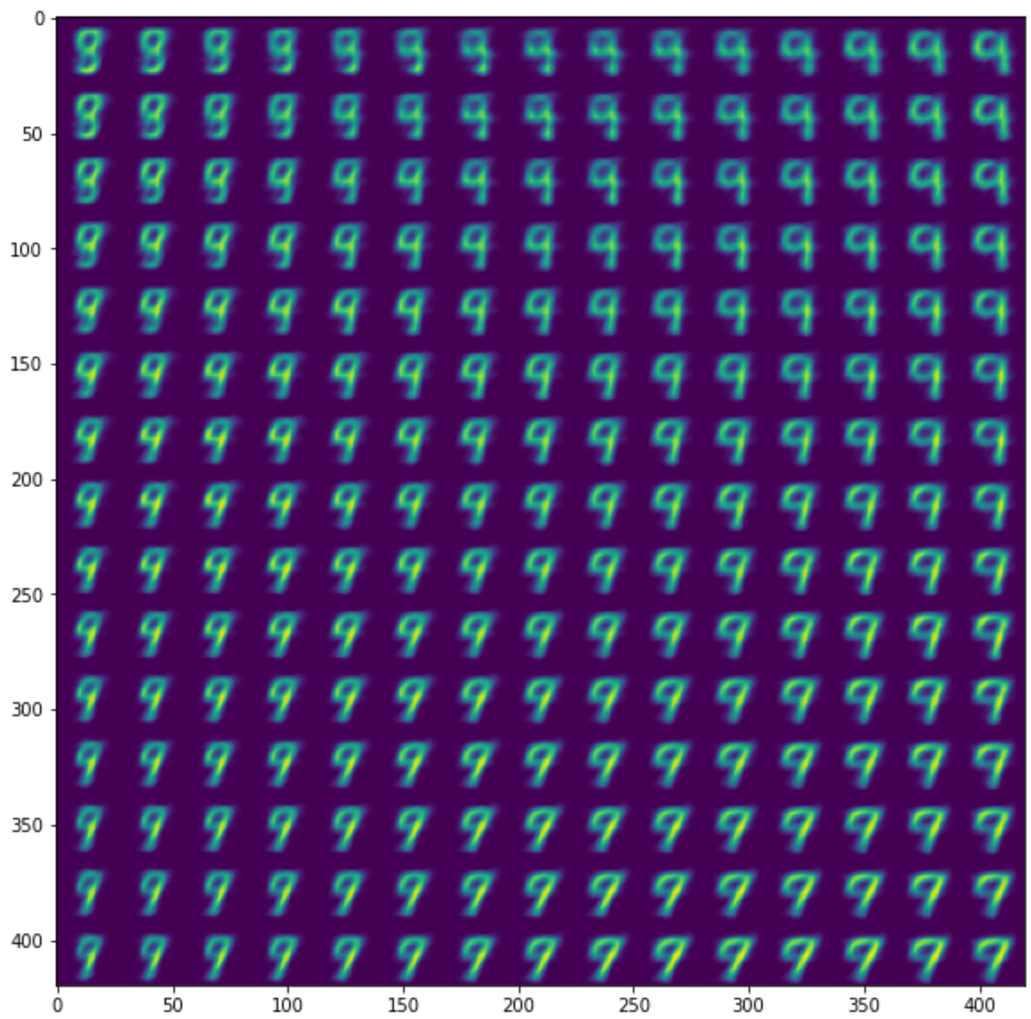
```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
```

```
history = vae.fit(x_train, x_train,
            epochs=100,
            batch_size=32,
            validation_data=(x_test, x_test))
```

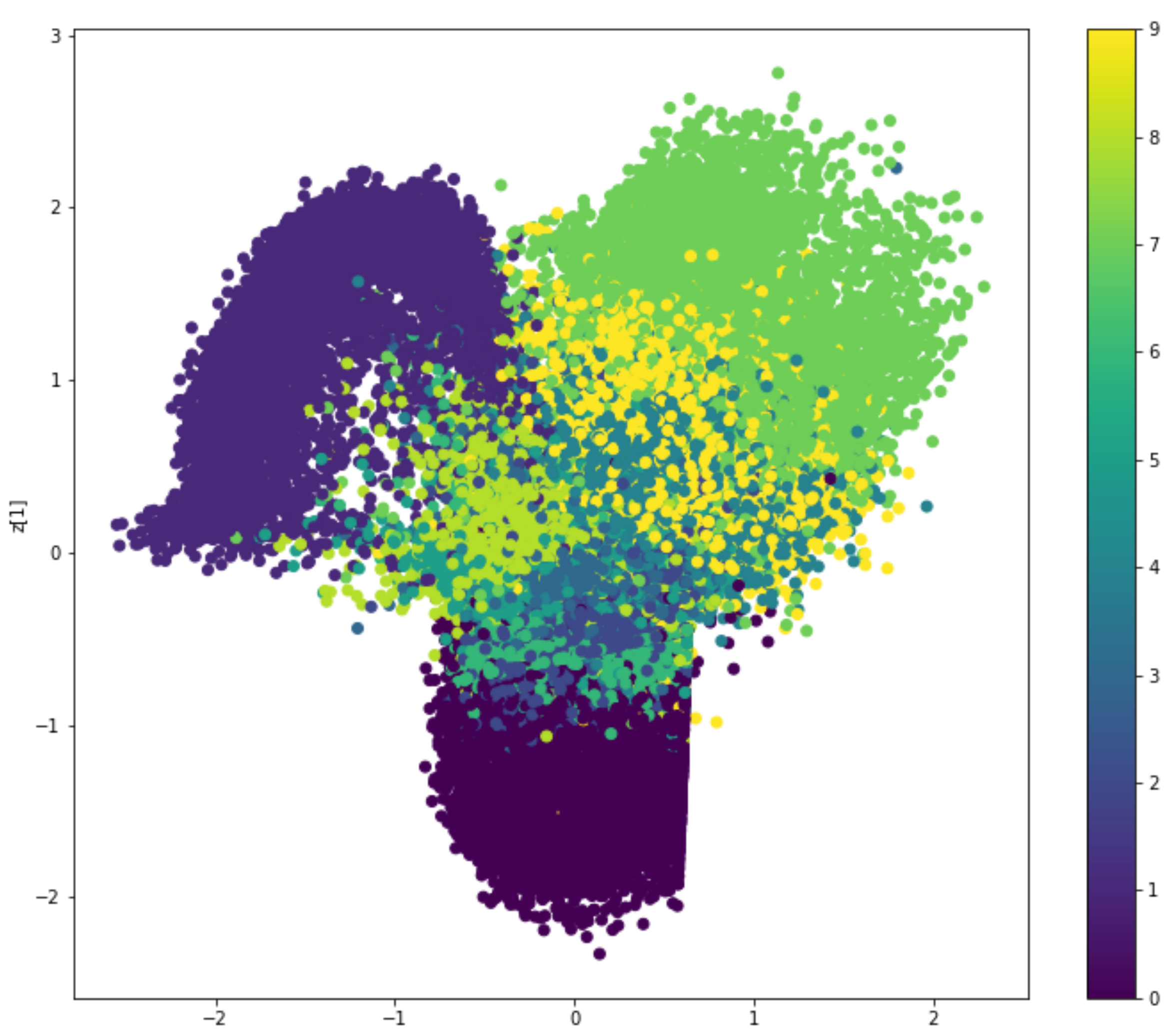
```
Epoch 7/100
1875/1875 [==============================] - 5s 3ms/step - loss: 163.2380 - val_loss: 162.9973
Epoch 8/100
1875/1875 [==============================] - 5s 3ms/step - loss: 162.6047 - val_loss: 162.3633
Epoch 9/100
1875/1875 [==============================] - 5s 3ms/step - loss: 162.0497 - val_loss: 162.0303
Epoch 10/100
1875/1875 [==============================] - 5s 3ms/step - loss: 161.6148 - val_loss: 161.6602
Epoch 11/100
1875/1875 [==============================] - 5s 3ms/step - loss: 161.2224 - val_loss: 161.0527
Epoch 12/100
1875/1875 [==============================] - 5s 3ms/step - loss: 160.8669 - val_loss: 160.8416
Epoch 13/100
1875/1875 [==============================] - 5s 3ms/step - loss: 160.5804 - val_loss: 160.6265
Epoch 14/100
1875/1875 [==============================] - 5s 3ms/step - loss: 160.2865 - val_loss: 160.3316
Epoch 15/100
1875/1875 [==============================] - 5s 3ms/step - loss: 160.0310 - val_loss: 160.4663
Epoch 16/100
1875/1875 [==============================] - 5s 3ms/step - loss: 159.8016 - val_loss: 159.8836
Epoch 17/100
1875/1875 [==============================] - 5s 3ms/step - loss: 159.5677 - val_loss: 159.7931
Epoch 18/100
1875/1875 [==============================] - 5s 3ms/step - loss: 159.3897 - val_loss: 159.5446
Epoch 19/100
1875/1875 [==============================] - 5s 3ms/step - loss: 159.1997 - val_loss: 159.3846
Epoch 20/100
1875/1875 [==============================] - 5s 3ms/step - loss: 159.0454 - val_loss: 159.1410
Epoch 21/100
1875/1875 [==============================] - 5s 3ms/step - loss: 158.8984 - val_loss: 159.1607
Epoch 22/100
1875/1875 [==============================] - 5s 3ms/step - loss: 158.7330 - val_loss: 158.9807
Epoch 23/100
1875/1875 [==============================] - 5s 3ms/step - loss: 158.6151 - val_loss: 158.8417
Epoch 24/100
1875/1875 [==============================] - 5s 3ms/step - loss: 158.4575 - val_loss: 158.7086
Epoch 25/100
1875/1875 [==============================] - 5s 3ms/step - loss: 158.3021 - val_loss: 158.5340
Epoch 26/100
1875/1875 [==============================] - 5s 3ms/step - loss: 158.0411 - val_loss: 158.4472
Epoch 27/100
1875/1875 [==============================] - 5s 3ms/step - loss: 157.8929 - val_loss: 158.5801
Epoch 28/100
1875/1875 [==============================] - 5s 3ms/step - loss: 157.7749 - val_loss: 158.1187
Epoch 29/100
1875/1875 [==============================] - 5s 3ms/step - loss: 157.6714 - val_loss: 157.9538
Epoch 30/100
1875/1875 [==============================] - 5s 3ms/step - loss: 157.5461 - val_loss: 157.8703
Epoch 31/100
1875/1875 [==============================] - 5s 3ms/step - loss: 157.4316 - val_loss: 157.9017
Epoch 32/100
1875/1875 [==============================] - 5s 3ms/step - loss: 157.3412 - val_loss: 157.6841
Epoch 33/100
1875/1875 [==============================] - 5s 3ms/step - loss: 157.2496 - val_loss: 157.5634
Epoch 34/100
1875/1875 [==============================] - 5s 3ms/step - loss: 157.1559 - val_loss: 157.5600
Epoch 35/100
1875/1875 [==============================] - 5s 3ms/step - loss: 157.0455 - val_loss: 157.3994
Epoch 36/100
1875/1875 [==============================] - 5s 3ms/step - loss: 156.9850 - val_loss: 157.6485
```

```
# Display a 2D manifold of the digits
n = 15  # figure with 15x15 digits
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
grid_x = np.linspace(0.05, 0.95, n)
grid_y = np.linspace(0.05, 0.95, n)

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]])
```

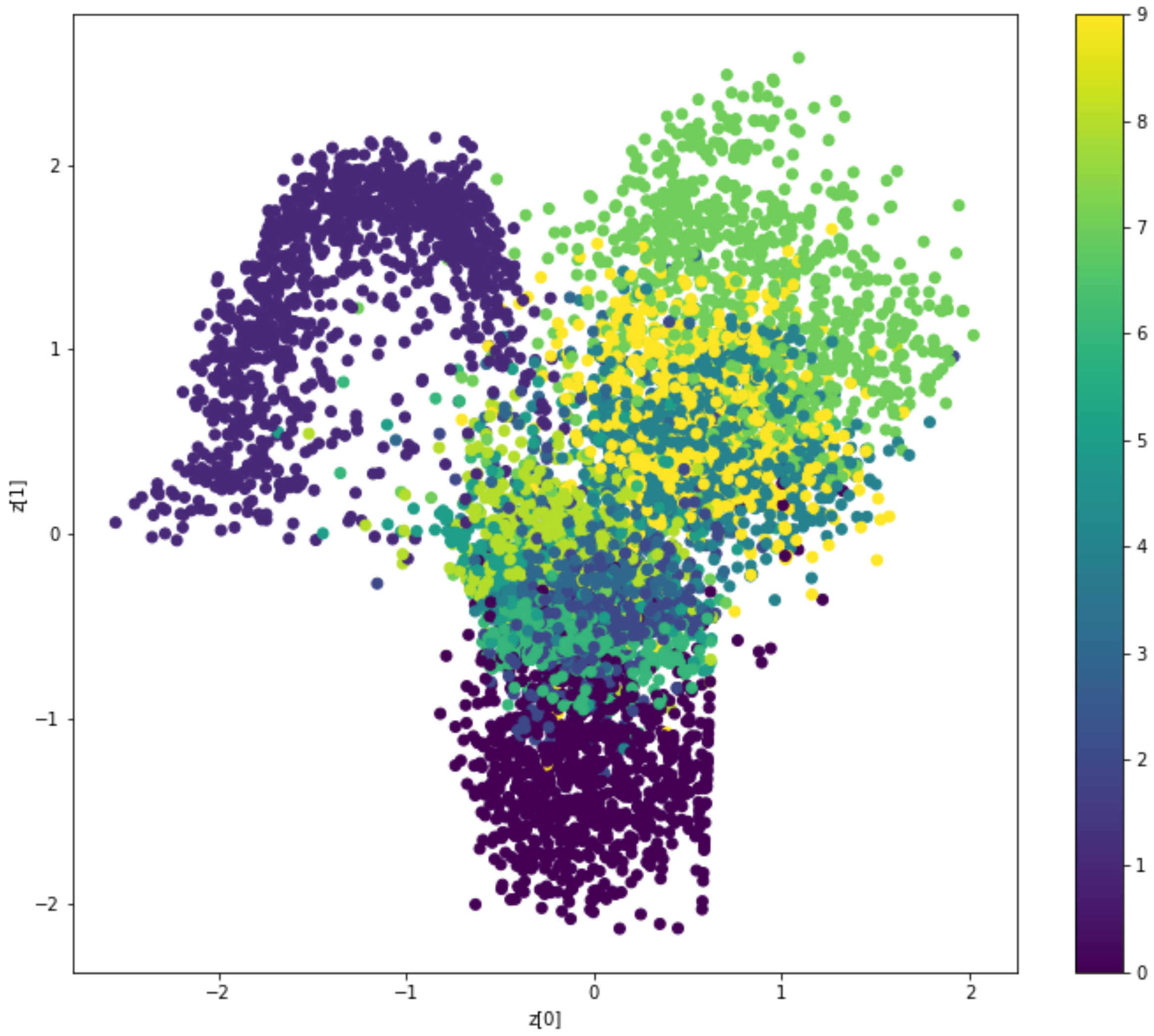
```
        x_decoded = decoder.predict(z_sample)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure)
plt.show()
```




```
plot_label_clusters(encoder, decoder, x_train, y_train)
```




```
plot_label_clusters(encoder, decoder, x_test, y_test)
```


z[0]
z[1]
−2
−1
0
1
2
0
1
2
3
4
5
6
7
8
9