University of Missouri-Kansas City

School of Computing and Engineering

ICP: ICP-07

Course Name: Big Data Analytics and Applications
Course ID: COMP-SCI 5542

Semester /Session : Spring 2022

Student ID : **Student Name**

16334245 Mustavi Islam

16321217 Keenan Flynn

Description of the Problem :
Variational Auto-Encoders (VAE) are a generational model used to generate new samples from a given set of training data The input and output of VAEs are the same. VAEs consist of two processes: encoding and decoding. During the encoding process, an input is compressed to lower dimensions. These lower dimensions are a coded representation of the input. During the decoding process the lower dimensions are reconstructed into the output. The output is a new piece of data that is similar to the training set but is distinct.

In this ICP, we will implement a VAE on the MNIST dataset to produce new handwritten digits.

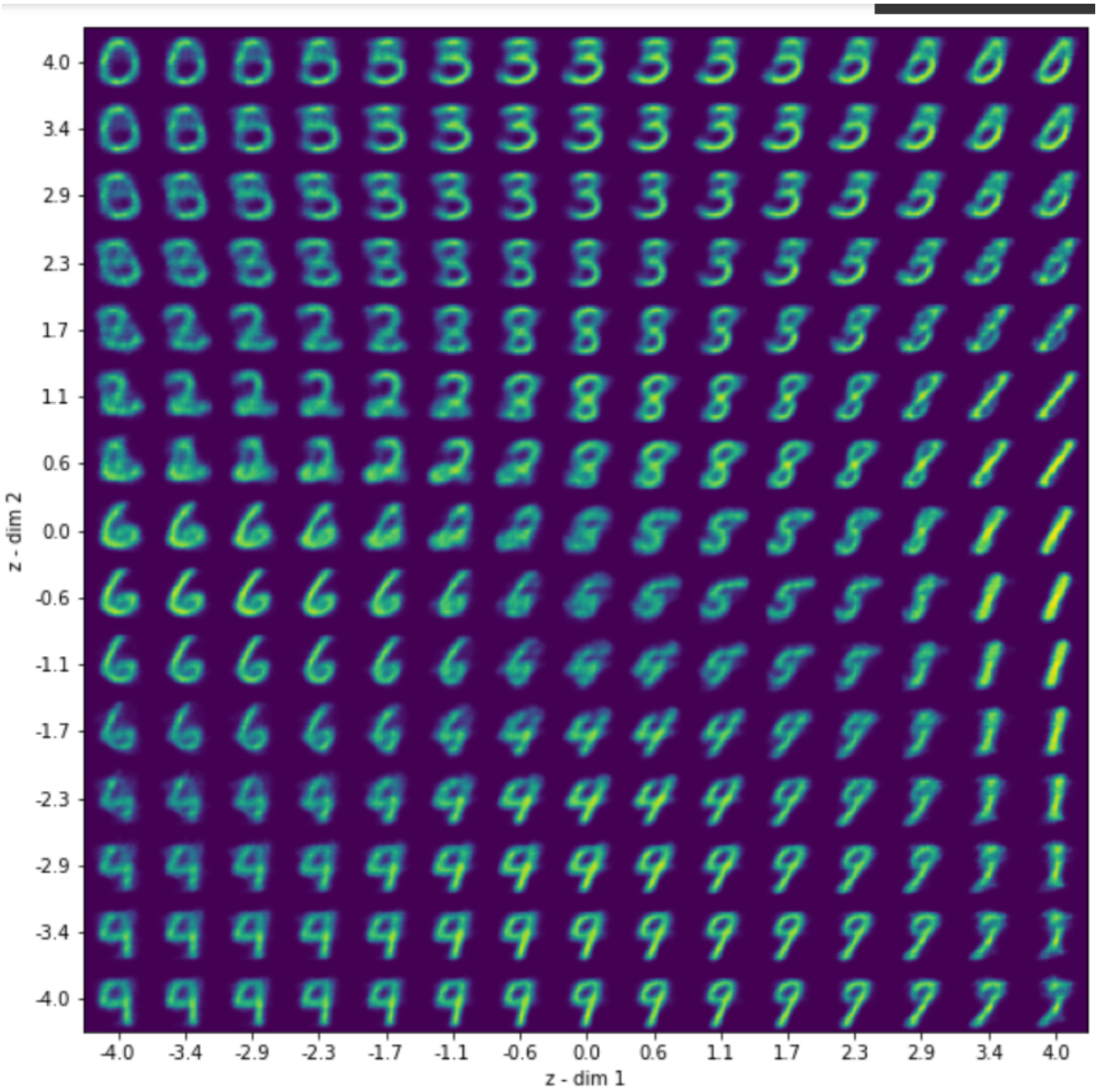
Description of Solution :
In this ICP, we added 8 layers. We added 2 Conv2D() in the encoder and 2 Conv2DTranspose() layers in the decoder. We added 4 BatchNormalization() layers after each of the added layers. We made this decision because the additional Conv2D() layers work to further encode the data. The Conv2DTranspose() are the inverse of the Conv2D() layers. By doing this, the code between the encoder and decoder has less dimensions and thus the output will hopefully have more variation. We are ultimately using the VAE to produce new samples so more variation will make the new samples distinct. Here is a step-by-step walkthrough of our method.

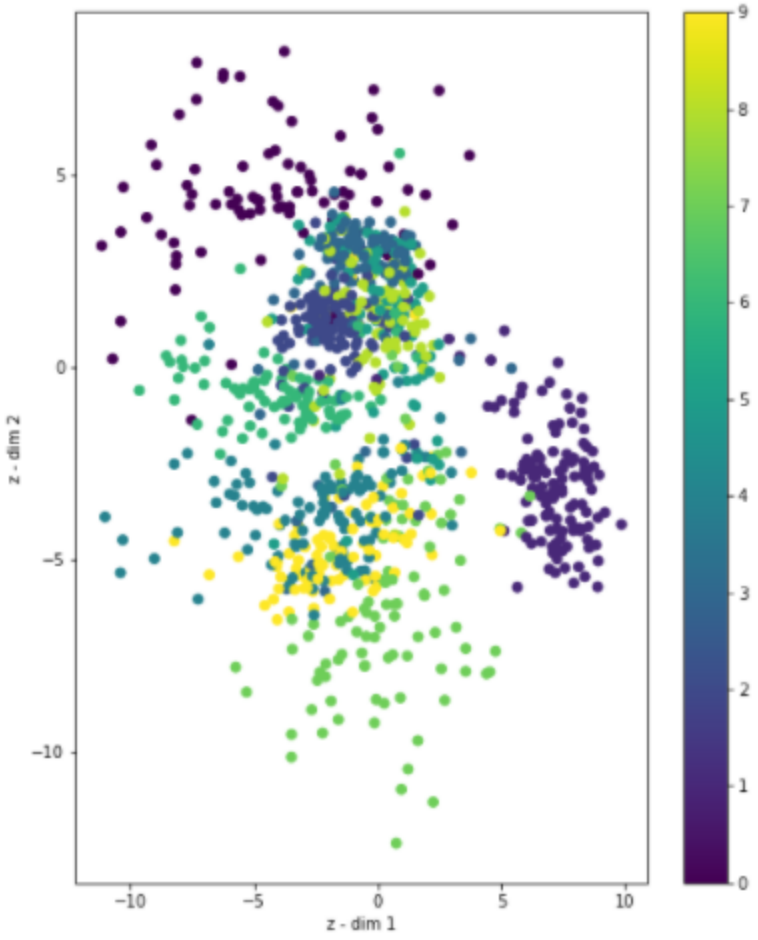
1. Get the dataset
    a. The MNIST dataset comes with the Keras library so we can import it through the mnist.load() method.
    b. Some exploration and visualization was done on the dataset. The training and testing dimensions were found and the first 10 handwritten digits were visualized.
2. Preprocessing
    a. The pixel strengths were normalized to floats so that they are in the range of (0,1).
    b. Standard values for number of epochs, batch size, test-train split, verbosity were defined.
    c. The input dimensions for the VAE were defined by getting the shape and number of channels of the training and testing set.
3. Encoder Definition
    a. The encoder reduces the dimensions of the input.
    b. The encoder has an input layer that defines the shape of the input.
    c. The encoder has 4 convolutional layers. These layers aggregate the input and thus reduce the dimensions. This reduction of dimensions is what enables VAEs to then construct accurate replications of the input.

        i. **We added 2 Conv2D() layers. The first Conv2D() layer has 32 filters and the second has 64() filters.**
    d. After every Conv2D() layer, we use a BatchNormilzation() layer. This layer normalizes the data so that the mean is 1 and the std is 0.
        i. **We added 2 BatchNormalization layers after our previously added Conv2D() layers.**
    e. After the convolutional and batch normalization layers, we use a Flatten() layer to get the 2D data into a 1D vector before passing that vector into a Dense() with 20 neurons.
    f. Finally the encoder passes the data into another BatchNormailization() layer before another Dense() layer with 2 neurons which is the latent dimensional space.
4. Defining Loss Function
    a. The VAE does not have an inherent loss function. We can reparametrize the samples into the form $\mu+\sigma 2\times\epsilon$. This will allow the model to learn and backpropagate.
5. Decoder Definition
    a. The Decoder is the inverse of the encoder. The decoder takes a low dimensional code and reconstructs an image similar to the input.
    b. Similar to the encoder, there is an input layer that defines the size of the input.
    c. A Dense() layer is added so that the output of the Input() layer can be transformed into exactly the same dimensions of what we had when we used the convolutional layers in the encoder.
    d. The output of the Dense() layer is then normalized and unFlattened using a Reshape() layer.
    e. Because the encoder has 4 convolutional layers, the decoder must have 4 inverse convolutional layers. These layers are called Conv2DTranspose() layers.
        i. **We added 2 Conv2DTranspose() layers in the decoder. 1 of these layers has 32 filters and the other has 64 filters.**
    f. Similar to the encoder, we also need BatchNormalization() layers after each convolutional layer.
        i. **We added 2 BatchNormalization() layers after the additional Conv2DTranspose() layers.**
    g. Finally we have an output layer which is an additional Conv2DTranspose() layer with Sigmoid activation and number of filters equal to the number of channels which is 2. A Sigmoid activation allows for a binary crossentropy loss function.
6. Model Compilation

a. The VAE model is constructed by combining the encoder and decoder.
b. We use binary cross entropy as the loss function and adam as the optimizer.
c. The model is fit with our predefined hyperparameters. The loss decreases to a value of 0.19.

7. Results
    a. Here is the visualization of our results.





Challenges:
One of the biggest challenges of this ICP was fully understanding the source code. This was our first time using the Keras Functional API. The Functional API is more flexible than the Sequential Model but also slightly more complex. We both studied this API before making any changes.

Another challenge we faced was understanding how the loss function works. For a gradient descent problem, we need a differentiable loss function. We followed the example code to reparametrize the input to the loss function so that we can use it to learn. While the specifics are highly mathematical, the team was able to understand why we did this process.

Learning Outcomes :

1. We learned how to use the Keras Functional API.
2. We learned how to create and modify a Keras Variational Auto-Encoder.
3. We learned how VAE using convolutional layers can be used for image problems.
4. We learned how to set up a loss function for a VAE.

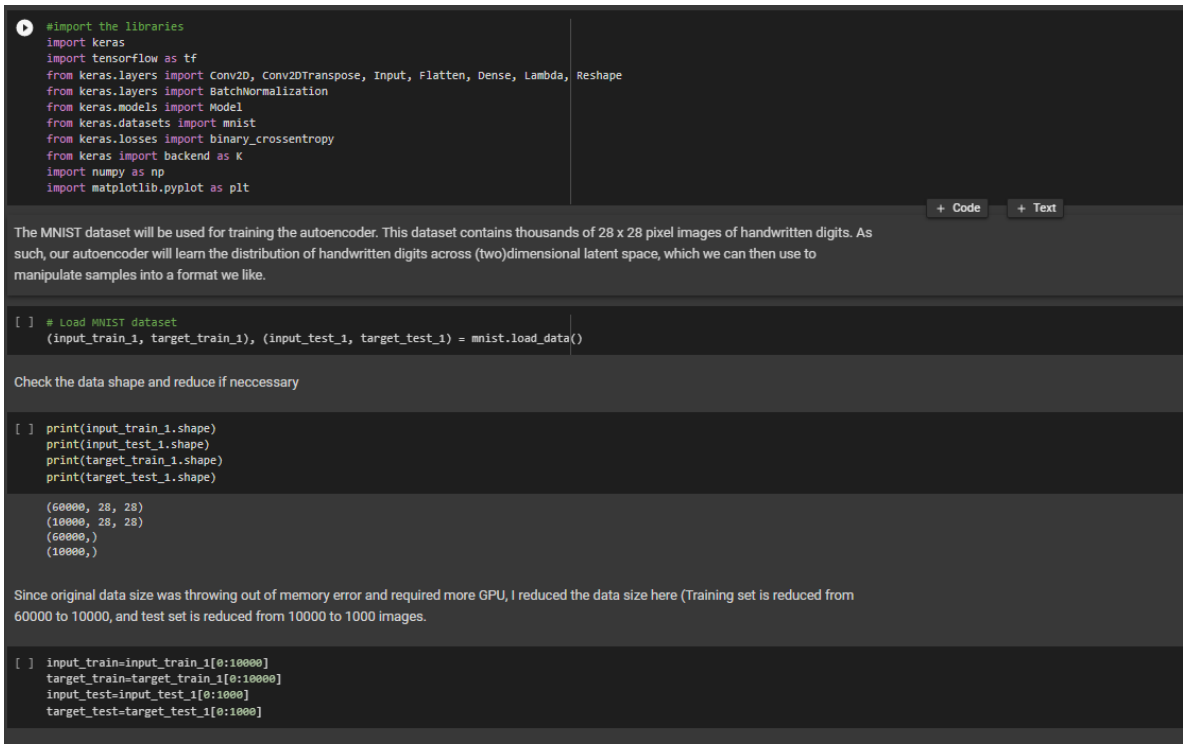
Resources:

https://github.com/christianversloot/machine-learning-articles/blob/main/how-to-create-a-variational-autoencoder-with-keras.md

Used for reference

Video Link: https://youtu.be/ZQbZbDTqXd8

Screenshots :



```
#import the libraries
import keras
import tensorflow as tf
from keras.layers import Conv2D, Conv2DTranspose, Input, Flatten, Dense, Lambda, Reshape
from keras.layers import BatchNormalization
from keras.models import Model
from keras.datasets import mnist
from keras.losses import binary_crossentropy
from keras import backend as K
import numpy as np
import matplotlib.pyplot as plt
```

+ Code + Text

The MNIST dataset will be used for training the autoencoder. This dataset contains thousands of 28 x 28 pixel images of handwritten digits. As such, our autoencoder will learn the distribution of handwritten digits across (two)dimensional latent space, which we can then use to manipulate samples into a format we like.

```
[ ] # Load MNIST dataset
    (input_train_1, target_train_1), (input_test_1, target_test_1) = mnist.load_data()
```

Check the data shape and reduce if neccessary

```
[ ] print(input_train_1.shape)
    print(input_test_1.shape)
    print(target_train_1.shape)
    print(target_test_1.shape)
```

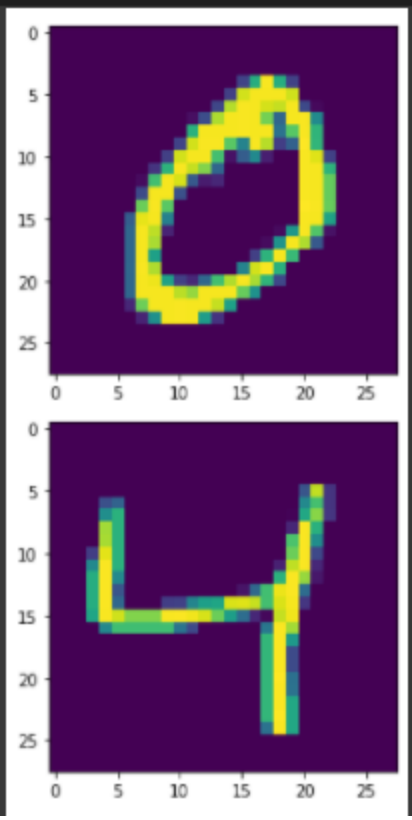
```
(60000, 28, 28)
(10000, 28, 28)
(60000,)
(10000,)
```

Since original data size was throwing out of memory error and required more GPU, I reduced the data size here (Training set is reduced from 60000 to 10000, and test set is reduced from 10000 to 1000 images.

```
[ ] input_train=input_train_1[0:10000]
    target_train=target_train_1[0:10000]
    input_test=input_test_1[0:1000]
    target_test=target_test_1[0:1000]
```

Print the first 10 images in training set

```
for i in range(10):
  pr_image = input_train[i]
  pr_image = np.array(pr_image, dtype='float')
  pixels = pr_image.reshape((28, 28))
  plt.imshow(pixels)
  plt.show()
```



Check the size of reduced data set

```
print(input_train.shape)
print(input_test.shape)
print(target_train.shape)
print(target_test.shape)
```

```
(10000, 28, 28)
(1000, 28, 28)
(10000,)
(1000,)
```

Model configuration: Setting config parameters for data and model.

The width and height of our configuration settings is determined by the training data. In our case, they will be img_width = img_height = 28, as the MNIST dataset contains samples that are 28 x 28 pixels.

Batch size is set to 128 samples per (mini)batch, which is quite normal. The same is true for the number of epochs, which was set to 50. 20% of the training data is used for validation purposes. This is also quite normal. Nothing special here.

Verbosity mode is set to True (by means of 1), which means that all the output is shown on screen.

The final two configuration settings are of relatively more interest. First, the latent space will be two-dimensional . Finally, the num_channels parameter can be configured to equal the number of image channels: for RGB data, it's 3 (red – green – blue), and for grayscale data (such as MNIST), it's 1

```
# Data & model configuration
img_width, img_height = input_train.shape[1], input_train.shape[2]
batch_size = 128
no_epochs = 50
validation_split = 0.2
verbosity = 1
latent_dim = 2
num_channels = 1
```

Next, we reshape the data so that it takes the shape (X, 28, 28, 1), where X is the number of samples in either the training or testing dataset. We also set (28, 28, 1) as input_shape.

Next, we parse the numbers as floats, which presumably speeds up the training process, and normalize it, which the neural network appreciates

```
# Reshape data
input_train = input_train.reshape(input_train.shape[0], img_height, img_width, num_channels)
input_test = input_test.reshape(input_test.shape[0], img_height, img_width, num_channels)
input_shape = (img_height, img_width, num_channels)

# Parse numbers as floats
input_train = input_train.astype('float32')
input_test = input_test.astype('float32')

# Normalize data
input_train = input_train / 255
input_test = input_test / 255
```

Creating the encoder

Now, it's time to create the encoder. This is a three-step process: firstly, we define it. Secondly, we perform something that is known as the reparameterization trick in order to allow us to link the encoder to the decoder later, to instantiate the VAE as a whole. But before that, we instantiate the encoder first, as our third and final step.

Encoder definition

The first step in the three-step process is the definition of our encoder. Following the connection process of the Keras Functional API, we link the layers together:

```
[ ] # Encoder Definition
    i       = Input(shape=input_shape, name='encoder_input')
    #COnv2D with 8 filters
    cx      = Conv2D(filters=8, kernel_size=3, strides=2, padding='same', activation='relu')(i)
    cx      = BatchNormalization()(cx)
    #COnv2D with 16 filters
    cx      = Conv2D(filters=16, kernel_size=3, strides=2, padding='same', activation='relu')(cx)
    cx      = BatchNormalization()(cx)
    #COnv2D with 32 filters
    cx      = Conv2D(filters=32, kernel_size=3, strides=2, padding='same', activation='relu')(i)
    cx      = BatchNormalization()(cx)
    #COnv2D with 64 filters
    cx      = Conv2D(filters=64, kernel_size=3, strides=2, padding='same', activation='relu')(cx)
    cx      = BatchNormalization()(cx)
    x       = Flatten()(cx)
    x       = Dense(20, activation='relu')(x)
    x       = BatchNormalization()(x)
    mu      = Dense(latent_dim, name='latent_mu')(x)
    sigma   = Dense(latent_dim, name='latent_sigma')(x)
```

Let's now take a look at the individual lines of code in more detail.

The first layer is the Input layer. It accepts data with input_shape = (28, 28, 1) and is named encoder_input.

Next up is a two-dimensional convolutional layer, or Conv2D in Keras terms. It learns 8 filters by deploying a 3 x 3 kernel which it convolves over the input. It has a stride of two which means that it skips over the input during the convolution as well, speeding up the learning process. It employs 'same' padding and ReLU activation. Do note that officially, it's best to use He init with ReLU activating layers. However, since the dataset is relatively small, it shouldn't be too much of a problem if you don't.

Subsequently, we use Batch Normalization. This layer ensures that the outputs of the Conv2D layer that are input to the next Conv2D layer have a steady mean and variance, likely $\mu=0.0,\sigma=1.0$ (plus some ϵ, an error term to ensure numerical stability). This benefits the learning process.

Once again, a Conv2D layer. It learns 16 filters and for the rest is equal to the first Conv2D layer.

BatchNormalization once more.

Next up, a Flatten layer, it only serves to flatten the multidimensional data from the convolutional layers into one-dimensional shape. This has to be done because the densely-connected layers that we use next require data to have this shape.

The next layer is a Dense layer with 20 output neurons. It's the autoencoder bottleneck we've been talking about.

BatchNormalization once more.

The next two layers, mu and sigma, are actually not separate from each other – look at the previous layer they are linked to (both x, i.e. the Dense(20) layer). The first outputs the mean values μ of the encoded input and the second one outputs the stddevs σ. With these, we can sample the random variables that constitute the point in latent space onto which some input is mapped. That's for the layers of our encoder.

The next step is to retrieve the shape of the final Conv2D output. We'll need it when defining the layers of our decoder

```
[ ]  # Get Conv2D shape for Conv2DTranspose operation in decoder
     conv_shape = K.int_shape(cx)
```

I'll try to explain the need for reparameritization briefly:

If you use neural networks (or, to be more precise, gradient descent) for optimizing the variational autoencoder, you effectively minimize some expected loss value, which can be estimated with Monte-Carlo techniques (Huang, n.d.). However, this requires that the loss function is differentiable, which is not necessarily the case, because it is dependent on the parameter of some probability distribution that we don't know about. In this case, it's possible to rewrite the equation, but then it no longer has the form of an expectation, making it impossible to use the Monte-Carlo techniques usable before.

However, if we can reparameterize the sample fed to the function into the shape $\mu+\sigma 2\times\epsilon$, it now becomes possible to use gradient descent for estimating the gradients accurately (Gunderson, n.d.; Huang, n.d.).

And that's precisely what we'll do in our code. We "sample" the value for z from the computed μ and σ values by resampling into mu + K.exp(sigma / 2) * eps.

```
[ ]  # Define sampling with reparameterization trick
     def sample_z(args):
       mu, sigma = args
       batch     = K.shape(mu)[0]
       dim       = K.int_shape(mu)[1]
       eps       = K.random_normal(shape=(batch, dim))
       return mu + K.exp(sigma / 2) * eps
```

We then use this with a Lambda to ensure that correct gradients are computed during the backwards pass based on our values for mu and sigma:

```
[ ]  # Use reparameterization trick to ensure correct gradient
     z       = Lambda(sample_z, output_shape=(latent_dim, ), name='z')([mu, sigma])
```

Encoder instantiation:

Now, it's time to instantiate the encoder – taking inputs through input layer i, and outputting the values generated by the mu, sigma and z layers (i.e., the individual means and standard deviations, and the point sampled from the random variable represented by them):

```
# Instantiate encoder
encoder = Model(i, [mu, sigma, z], name='encoder')
encoder.summary()
```

```
Model: "encoder"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 encoder_input (InputLayer)     [(None, 28, 28, 1)]  0           []

 conv2d_6 (Conv2D)              (None, 14, 14, 32)   320         ['encoder_input[0][0]']

 batch_normalization_12 (BatchN  (None, 14, 14, 32)  128         ['conv2d_6[0][0]']
 ormalization)

 conv2d_7 (Conv2D)              (None, 7, 7, 64)     18496       ['batch_normalization_12[0][0]']

 batch_normalization_13 (BatchN  (None, 7, 7, 64)    256         ['conv2d_7[0][0]']
 ormalization)

 flatten_1 (Flatten)            (None, 3136)         0           ['batch_normalization_13[0][0]']

 dense_2 (Dense)                (None, 20)           62740       ['flatten_1[0][0]']

 batch_normalization_14 (BatchN  (None, 20)          80          ['dense_2[0][0]']
 ormalization)

 latent_mu (Dense)              (None, 2)            42          ['batch_normalization_14[0][0]']

 latent_sigma (Dense)           (None, 2)            42          ['batch_normalization_14[0][0]']

 z (Lambda)                     (None, 2)            0           ['latent_mu[0][0]',
                                                                  'latent_sigma[0][0]']

==================================================================================================
Total params: 82,104
Trainable params: 81,872
Non-trainable params: 232
__________________________________________________________________________________________________
```

Creating the decoder

Creating the decoder is a bit simpler and boils down to a two-step process: defining it, and instantiating it.

```
# Decoder Definition
d_i   = Input(shape=(latent_dim, ), name='decoder_input')
x     = Dense(conv_shape[1] * conv_shape[2] * conv_shape[3], activation='relu')(d_i)
x     = BatchNormalization()(x)
x     = Reshape((conv_shape[1], conv_shape[2], conv_shape[3]))(x)

#COnv2DTranspose with 64 filters
cx    = Conv2DTranspose(filters=64, kernel_size=3, strides=2, padding='same', activation='relu')(x)
cx    = BatchNormalization()(cx)
#COnv2DTranspose with 32 filters
cx    = Conv2DTranspose(filters=32, kernel_size=3, strides=2, padding='same',  activation='relu')(cx)
cx    = BatchNormalization()(cx)
#COnv2DTranspose with 16 filters
cx    = Conv2DTranspose(filters=16, kernel_size=3, strides=2, padding='same', activation='relu')(x)
cx    = BatchNormalization()(cx)
#COnv2DTranspose with 8 filters
cx    = Conv2DTranspose(filters=8, kernel_size=3, strides=2, padding='same',  activation='relu')(cx)
cx    = BatchNormalization()(cx)

o     = Conv2DTranspose(filters=num_channels, kernel_size=3, activation='sigmoid', padding='same', name='decoder_output')(cx)
```

Our decoder also starts with an Input layer, the decoder_input layer. It takes input with the shape (latent_dim,), which as we will see is the vector we sampled for z with our encoder.

If we'd like to upsample the point in latent space with Conv2DTranspose layers, in exactly the opposite symmetrical order as with we downsampled with our encoder, we must first bring back the data from shape (latent_dim,) into some shape that can be reshaped into the output shape of the last convolutional layer of our encoder.

This is why you needed the conv_shape variable. We'll thus now add a Dense layer which has conv_shape[1] * conv_shape[2] * conv_shape[3] output, and converts the latent space into many outputs.

We next use a Reshape layer to convert the output of the Dense layer into the output shape of the last convolutional layer: (conv_shape[1], conv_shape[2], conv_shape[3] = (7, 7, 16). Sixteen filters learnt with 7 x 7 pixels per filter.

We then use Conv2DTranspose and BatchNormalization in the exact opposite order as with our encoder to upsample our data into 28 x 28 pixels (which is equal to the width and height of our inputs). However, we still have 8 filters, so the shape so far is (28, 28, 8).

We therefore add a final Conv2DTranspose layer which does nothing to the width and height of the data, but ensures that the number of filters learns equals num_channels. For MNIST data, where num_channels = 1, this means that the shape of our output will be (28, 28, 1). This last layer also uses Sigmoid activation, which allows us to use binary crossentropy loss when computing the reconstruction loss part of our loss function.

Decoder instantiation

The next thing we do is instantiate the decoder:

It takes the inputs from the decoder input layer d_i and outputs whatever is output by the output layer o.

```
# Instantiate decoder
decoder = Model(d_i, o, name='decoder')
decoder.summary()
```

```
Model: "decoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 decoder_input (InputLayer)  [(None, 2)]               0

 dense_3 (Dense)             (None, 3136)              9408

 batch_normalization_15 (Bat  (None, 3136)             12544
 chNormalization)

 reshape_1 (Reshape)         (None, 7, 7, 64)          0

 conv2d_transpose_6 (Conv2DT  (None, 14, 14, 16)       9232
 ranspose)

 batch_normalization_18 (Bat  (None, 14, 14, 16)       64
 chNormalization)

 conv2d_transpose_7 (Conv2DT  (None, 28, 28, 8)        1160
 ranspose)

 batch_normalization_19 (Bat  (None, 28, 28, 8)        32
 chNormalization)

 decoder_output (Conv2DTrans  (None, 28, 28, 1)        73
 pose)

=================================================================
Total params: 32,513
Trainable params: 26,193
Non-trainable params: 6,320
_________________________________________________________________
```

```
# Instantiate VAE
vae_outputs = decoder(encoder(i)[2])
vae         = Model(i, vae_outputs, name='vae')
vae.summary()
```

```
Model: "vae"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder_input (InputLayer)  [(None, 28, 28, 1)]       0

 encoder (Functional)        [(None, 2),               82104
                              (None, 2),
                              (None, 2)]

 decoder (Functional)        (None, 28, 28, 1)         32513

=================================================================
Total params: 114,617
Trainable params: 108,065
Non-trainable params: 6,552
_________________________________________________________________
```

Compilation & training:

we can compile our model. We do so using the Adam optimizer and binary crossentropy loss function.

tf.config.run_functions_eagerly(True) is added to overcome the vae.fit error with colab, you may not need it if you are running the code in pycharm or jupyter notebooks.

```
tf.config.run_functions_eagerly(True)
# Compile VAE
vae.compile(optimizer='adam', loss='binary_crossentropy')

# Train autoencoder
vae.fit(input_train, input_train, epochs = no_epochs, batch_size = batch_size, validation_split = validation_split)
```

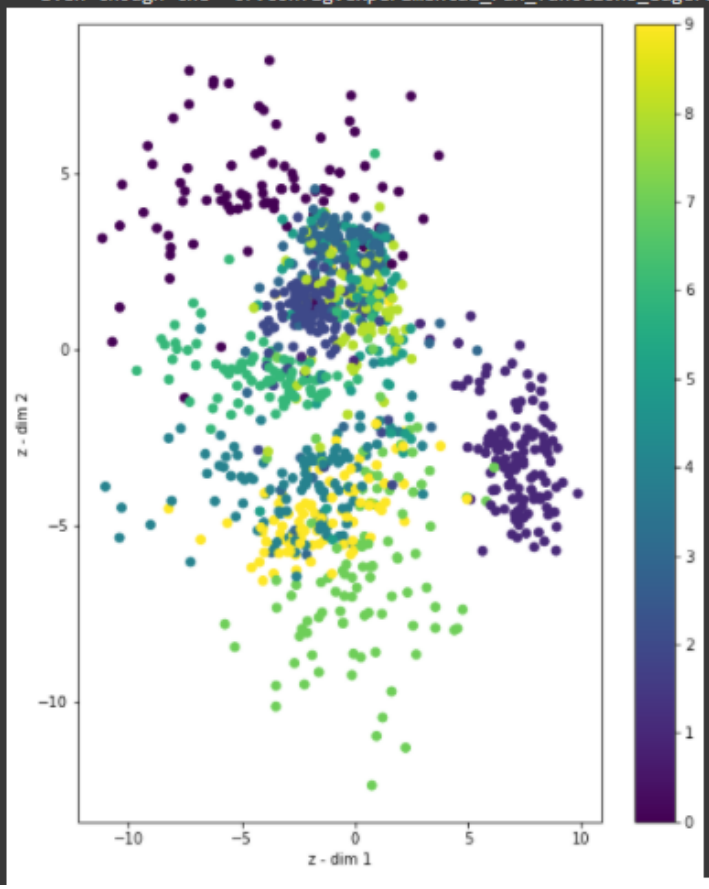
```
Epoch 1/50
 1/63 [..............................] - ETA: 5s - loss: 0.9125/usr/local/lib/python3.7/dist-packages/tensorflow/python/data/ops/structur
  "Even though the `tf.config.experimental_run_functions_eagerly` "
63/63 [==============================] - 3s 50ms/step - loss: 0.6195 - val_loss: 0.5400
Epoch 2/50
63/63 [==============================] - 3s 45ms/step - loss: 0.4037 - val_loss: 0.3943
Epoch 3/50
63/63 [==============================] - 3s 46ms/step - loss: 0.2771 - val_loss: 0.3356
Epoch 4/50
63/63 [==============================] - 3s 45ms/step - loss: 0.2440 - val_loss: 0.3038
Epoch 5/50
63/63 [==============================] - 3s 45ms/step - loss: 0.2335 - val_loss: 0.2771
Epoch 6/50
63/63 [==============================] - 3s 46ms/step - loss: 0.2268 - val_loss: 0.2594
Epoch 7/50
63/63 [==============================] - 3s 45ms/step - loss: 0.2218 - val_loss: 0.2426
Epoch 8/50
63/63 [==============================] - 3s 44ms/step - loss: 0.2181 - val_loss: 0.2324
Epoch 9/50
63/63 [==============================] - 3s 45ms/step - loss: 0.2156 - val_loss: 0.2239
Epoch 10/50
63/63 [==============================] - 3s 45ms/step - loss: 0.2117 - val_loss: 0.2191
Epoch 11/50
```

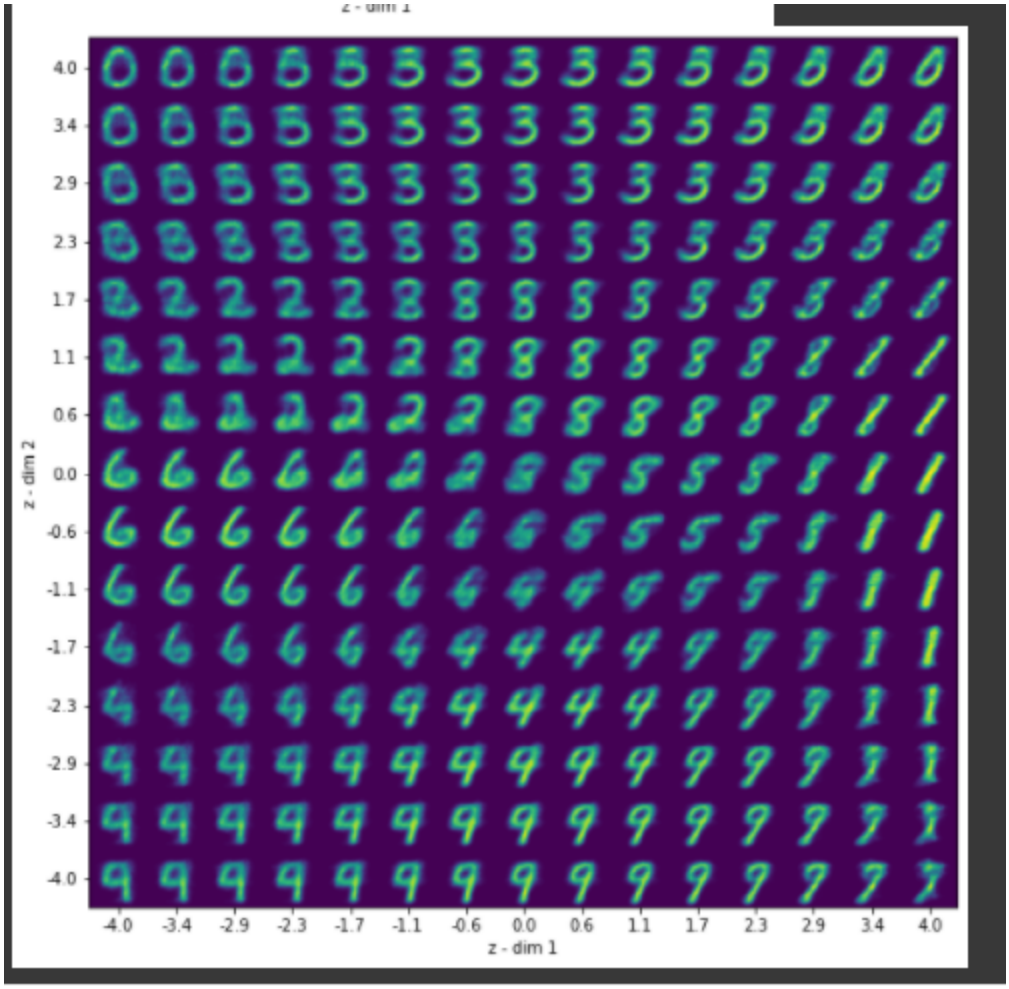
```
# Results visualization
# Credits for original visualization code: https://keras.io/examples/variational_autoencoder_deconv/


def viz_latent_space(encoder, data):
  input_data, target_data = data
  mu, _, _ = encoder.predict(input_data)
  plt.figure(figsize=(8, 10))
  plt.scatter(mu[:, 0], mu[:, 1], c=target_data)
  plt.xlabel('z - dim 1')
  plt.ylabel('z - dim 2')
  plt.colorbar()
  plt.show()
```

```
[ ] def viz_decoded(encoder, decoder, data):
      num_samples = 15
      figure = np.zeros((img_width * num_samples, img_height * num_samples, num_channels))
      grid_x = np.linspace(-4, 4, num_samples)
      grid_y = np.linspace(-4, 4, num_samples)[::-1]
      for i, yi in enumerate(grid_y):
          for j, xi in enumerate(grid_x):
              z_sample = np.array([[xi, yi]])
              x_decoded = decoder.predict(z_sample)
              digit = x_decoded[0].reshape(img_width, img_height, num_channels)
              figure[i * img_width: (i + 1) * img_width,
                     j * img_height: (j + 1) * img_height] = digit
      plt.figure(figsize=(10, 10))
      start_range = img_width // 2
      end_range = num_samples * img_width + start_range + 1
      pixel_range = np.arange(start_range, end_range, img_width)
      sample_range_x = np.round(grid_x, 1)
      sample_range_y = np.round(grid_y, 1)
      plt.xticks(pixel_range, sample_range_x)
      plt.yticks(pixel_range, sample_range_y)
      plt.xlabel('z - dim 1')
      plt.ylabel('z - dim 2')
      # matplotlib.pyplot.imshow() needs a 2D array, or a 3D array with the third dimension being of shape 3 or 4!
      # So reshape if necessary
      fig_shape = np.shape(figure)
      if fig_shape[2] == 1:
        figure = figure.reshape((fig_shape[0], fig_shape[1]))
      # Show image
      plt.imshow(figure)
      plt.show()
```

```
[ ]  # Plot results
     data = (input_test, target_test)
     viz_latent_space(encoder, data)
     viz_decoded(encoder, decoder, data)
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/data/ops/structured_function.py:265: UserWarning
  "Even though the `tf.config.experimental_run_functions_eagerly` "
```




z - dim 1
4.0
3.4
2.9
2.3
1.7
1.1
0.6
0.0
-0.6
-1.1
-1.7
-2.3
-2.9
-3.4
-4.0
z - dim 2
-4.0
-3.4
-2.9
-2.3
-1.7
-1.1
-0.6
0.0
0.6
1.1
1.7
2.3
2.9
3.4
4.0
z - dim 1