CS5542 BIG DATA APPS AND ANALYTICS

In Class Programming – 8

Variational Autoencoders

Kalyani Nikure
Kmn6bg@umkc.edu

Table of Contents

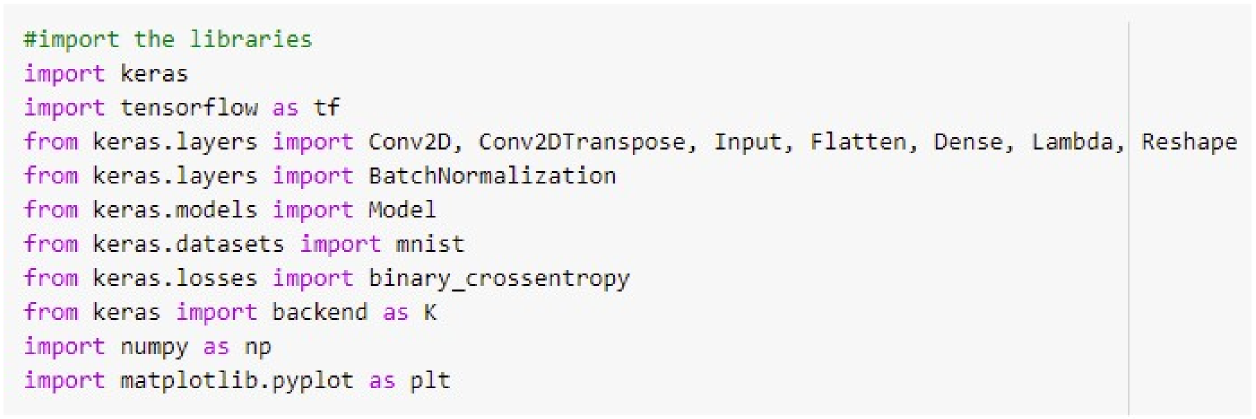
Description

Use the same data and source code but add two more layers to encoder path and their corresponding two layers to decoder path, run the new model and report your findings. In your report specify which 4 layers (2 layers in encoder path and 2 layers in decoder path) have you added and explain why you added those (their function).

Examples of layers that can be added Conv2D, Batchnorm, Conv2DTranspose etc.

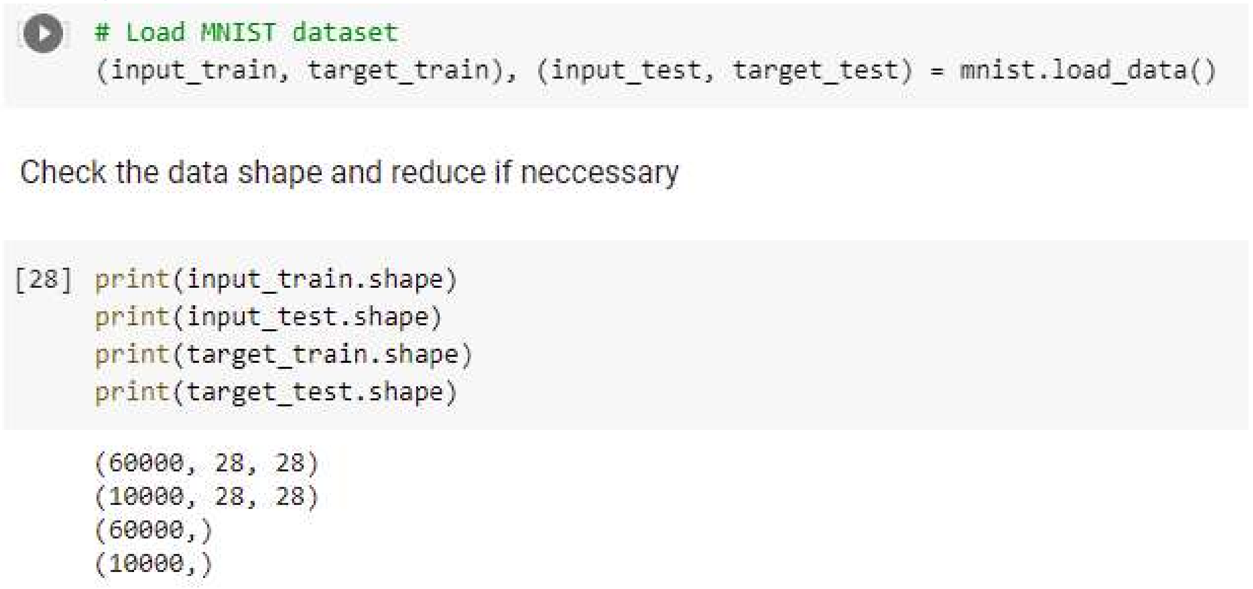
Detailed Steps Explanation

1. Import Models required



```
#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
```

2. Loading data



```
# Load MNIST dataset
(input_train, target_train), (input_test, target_test) = mnist.load_data()
```

Check the data shape and reduce if neccessary

```
[28] print(input_train.shape)
     print(input_test.shape)
     print(target_train.shape)
     print(target_test.shape)
```

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

3. Setting config parameters for data and model

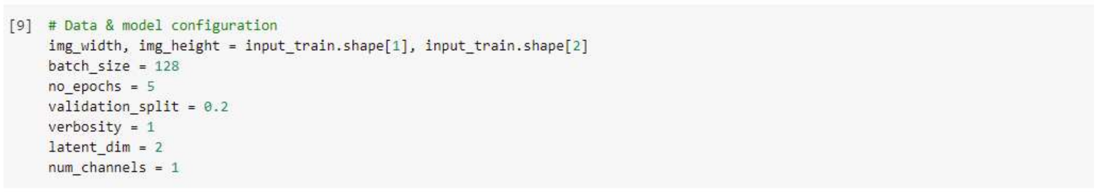
Model configuration: Setting config parameters for data and model.

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

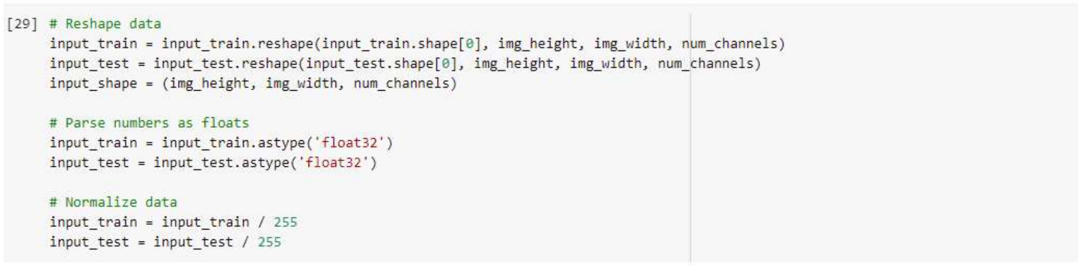


```
[9]  # Data & model configuration
     img_width, img_height = input_train.shape[1], input_train.shape[2]
     batch_size = 128
     no_epochs = 5
     validation_split = 0.2
     verbosity = 1
     latent_dim = 2
     num_channels = 1
```

4. Data preprocessing

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



```
[29] # 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
```

5. Encoder creation

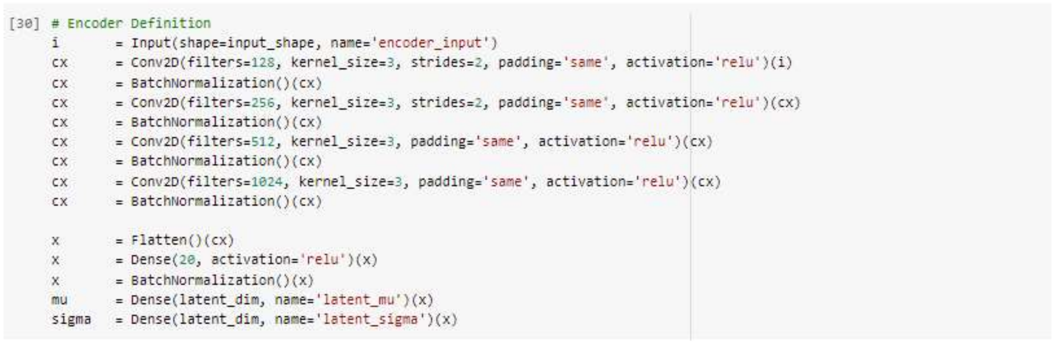
1. Encoder definition

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:



```
[30] # Encoder Definition
     i       = Input(shape=input_shape, name='encoder_input')
     cx      = Conv2D(filters=128, kernel_size=3, strides=2, padding='same', activation='relu')(i)
     cx      = BatchNormalization()(cx)
     cx      = Conv2D(filters=256, kernel_size=3, strides=2, padding='same', activation='relu')(cx)
     cx      = BatchNormalization()(cx)
     cx      = Conv2D(filters=512, kernel_size=3, padding='same', activation='relu')(cx)
     cx      = BatchNormalization()(cx)
     cx      = Conv2D(filters=1024, kernel_size=3, 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

2. Reparameterization trick

```
[31] # 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.).

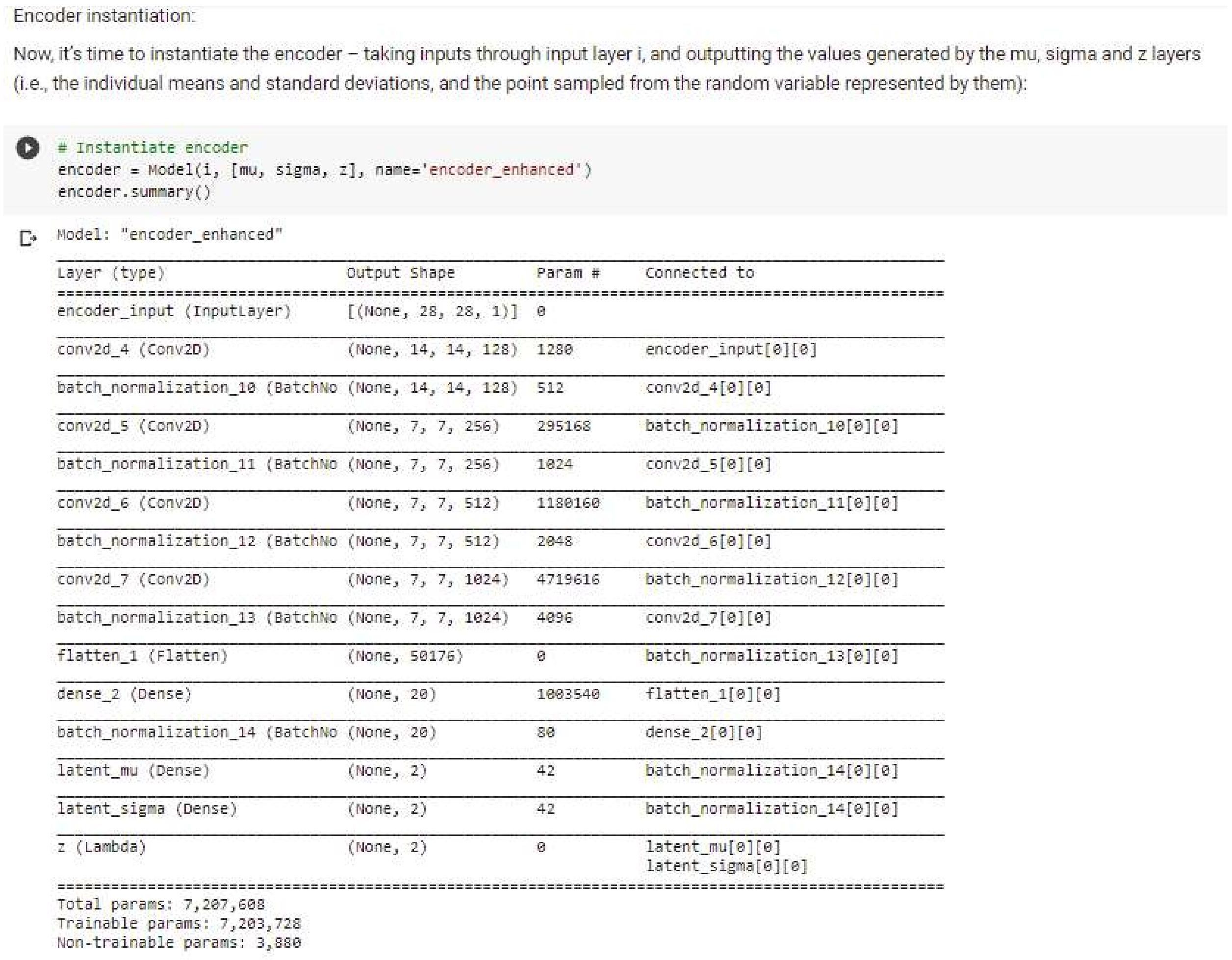
And that's precisely what we'll do in our code. We "sample" the value for z from the computed μ and σ values by resampling into mu + K.exp(sigma / 2) * eps.

```
[32] # 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:

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

3. Encoder instantiation



Encoder instantiation:

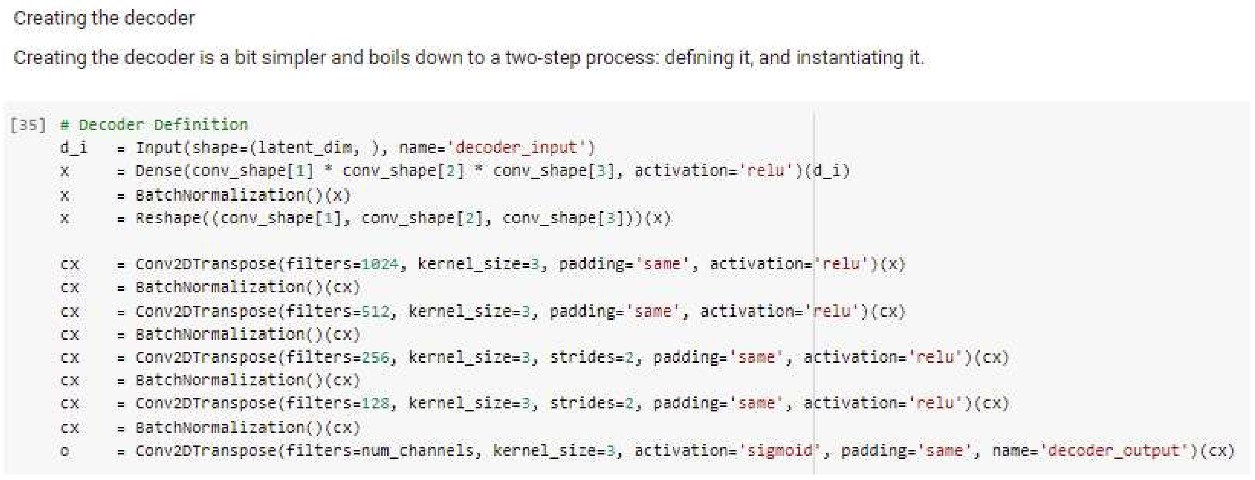
Now, it's time to instantiate the encoder – taking inputs through input layer i, and outputting the values generated by the mu, sigma and z layers (i.e., the individual means and standard deviations, and the point sampled from the random variable represented by them):

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

```
Model: "encoder_enhanced"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
encoder_input (InputLayer)      [(None, 28, 28, 1)]  0
__________________________________________________________________________________________________
conv2d_4 (Conv2D)               (None, 14, 14, 128)  1280        encoder_input[0][0]
__________________________________________________________________________________________________
batch_normalization_10 (BatchNo (None, 14, 14, 128)  512         conv2d_4[0][0]
__________________________________________________________________________________________________
conv2d_5 (Conv2D)               (None, 7, 7, 256)    295168      batch_normalization_10[0][0]
__________________________________________________________________________________________________
batch_normalization_11 (BatchNo (None, 7, 7, 256)    1024        conv2d_5[0][0]
__________________________________________________________________________________________________
conv2d_6 (Conv2D)               (None, 7, 7, 512)    1180160     batch_normalization_11[0][0]
__________________________________________________________________________________________________
batch_normalization_12 (BatchNo (None, 7, 7, 512)    2048        conv2d_6[0][0]
__________________________________________________________________________________________________
conv2d_7 (Conv2D)               (None, 7, 7, 1024)   4719616     batch_normalization_12[0][0]
__________________________________________________________________________________________________
batch_normalization_13 (BatchNo (None, 7, 7, 1024)   4096        conv2d_7[0][0]
__________________________________________________________________________________________________
flatten_1 (Flatten)             (None, 50176)        0           batch_normalization_13[0][0]
__________________________________________________________________________________________________
dense_2 (Dense)                 (None, 20)           1003540     flatten_1[0][0]
__________________________________________________________________________________________________
batch_normalization_14 (BatchNo (None, 20)           80          dense_2[0][0]
__________________________________________________________________________________________________
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: 7,207,608
Trainable params: 7,203,728
Non-trainable params: 3,880
```

6. Decoder creation

1. Decoder definition



Creating the decoder

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

```
[35] # 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)

     cx     = Conv2DTranspose(filters=1024, kernel_size=3, padding='same', activation='relu')(x)
     cx     = BatchNormalization()(cx)
     cx     = Conv2DTranspose(filters=512, kernel_size=3, padding='same', activation='relu')(cx)
     cx     = BatchNormalization()(cx)
     cx     = Conv2DTranspose(filters=256, kernel_size=3, strides=2, padding='same', activation='relu')(cx)
     cx     = BatchNormalization()(cx)
     cx     = Conv2DTranspose(filters=128, 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).

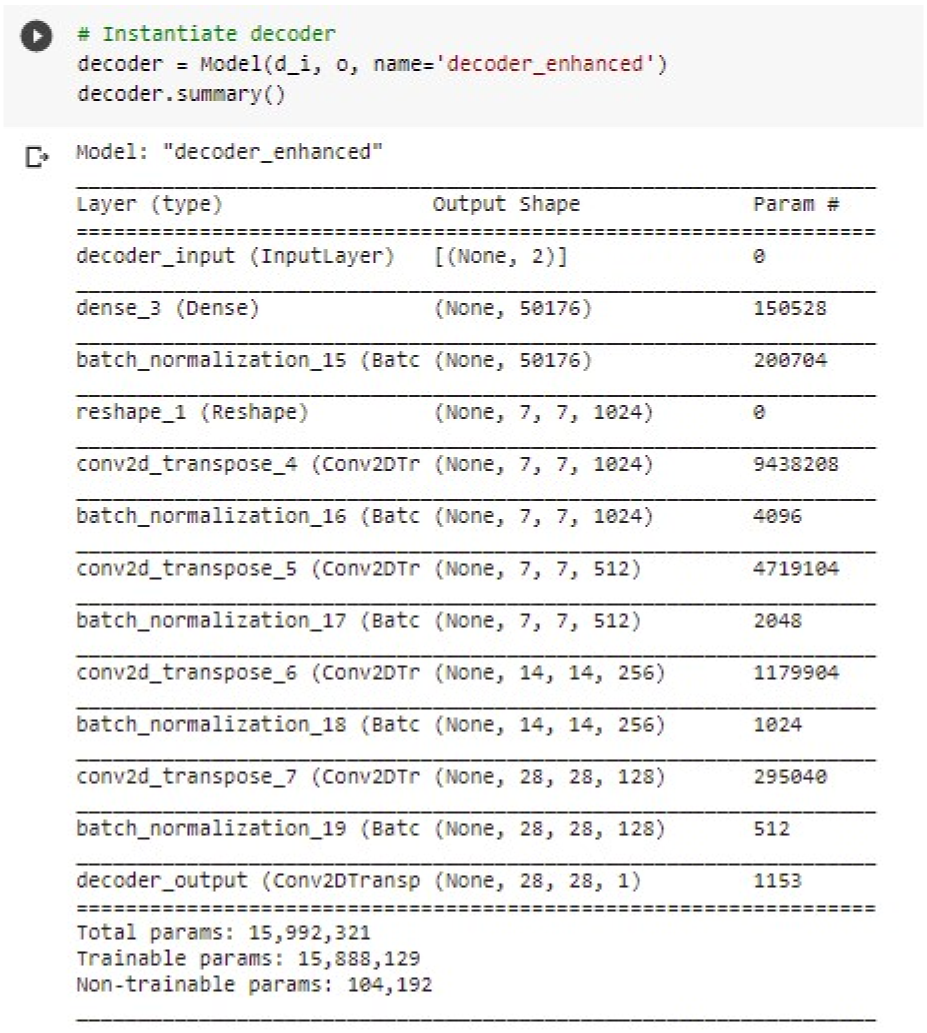
We therefore add a final Conv2DTranspose layer which does nothing to the width and height of the data, but ensures that the number of filters learns equals num_channels. For MNIST data, where num_channels = 1, this means that the shape of our output will be (28, 28, 1). This last layer also uses Sigmoid 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.

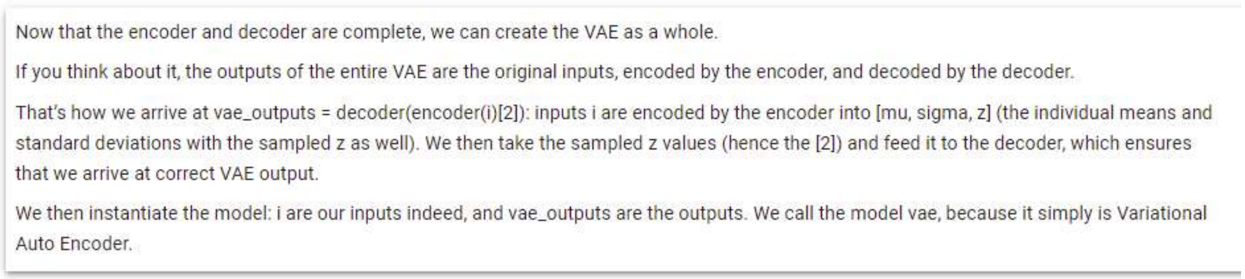
2. Decoder instantiation



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

```
Model: "decoder_enhanced"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
decoder_input (InputLayer)   [(None, 2)]               0
_________________________________________________________________
dense_3 (Dense)              (None, 50176)             150528
_________________________________________________________________
batch_normalization_15 (Batc (None, 50176)             200704
_________________________________________________________________
reshape_1 (Reshape)          (None, 7, 7, 1024)        0
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 7, 7, 1024)        9438208
_________________________________________________________________
batch_normalization_16 (Batc (None, 7, 7, 1024)        4096
_________________________________________________________________
conv2d_transpose_5 (Conv2DTr (None, 7, 7, 512)         4719104
_________________________________________________________________
batch_normalization_17 (Batc (None, 7, 7, 512)         2048
_________________________________________________________________
conv2d_transpose_6 (Conv2DTr (None, 14, 14, 256)       1179904
_________________________________________________________________
batch_normalization_18 (Batc (None, 14, 14, 256)       1024
_________________________________________________________________
conv2d_transpose_7 (Conv2DTr (None, 28, 28, 128)       295040
_________________________________________________________________
batch_normalization_19 (Batc (None, 28, 28, 128)       512
_________________________________________________________________
decoder_output (Conv2DTransp (None, 28, 28, 1)         1153
=================================================================
Total params: 15,992,321
Trainable params: 15,888,129
Non-trainable params: 104,192
_________________________________________________________________
```

7. Creating the whole VAE

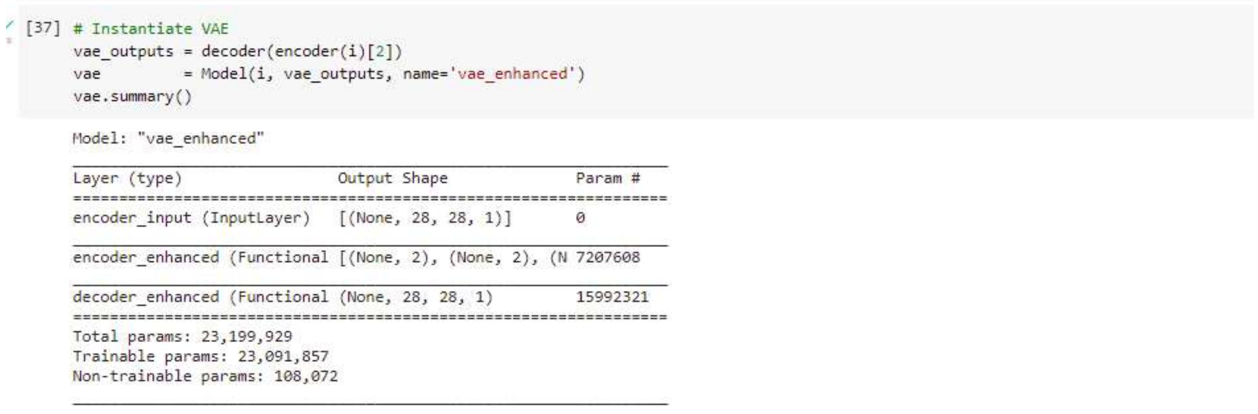


Now that the encoder and decoder are complete, we can create the VAE as a whole.

If you think about it, the outputs of the entire VAE are the original inputs, encoded by the encoder, and decoded by the decoder.

That's how we arrive at vae_outputs = decoder(encoder(i)[2]): inputs i are encoded by the encoder into [mu, sigma, z] (the individual means and standard deviations with the sampled z as well). We then take the sampled z values (hence the [2]) and feed it to the decoder, which ensures that we arrive at correct VAE output.

We then instantiate the model: i are our inputs indeed, and vae_outputs are the outputs. We call the model vae, because it simply is Variational Auto Encoder.



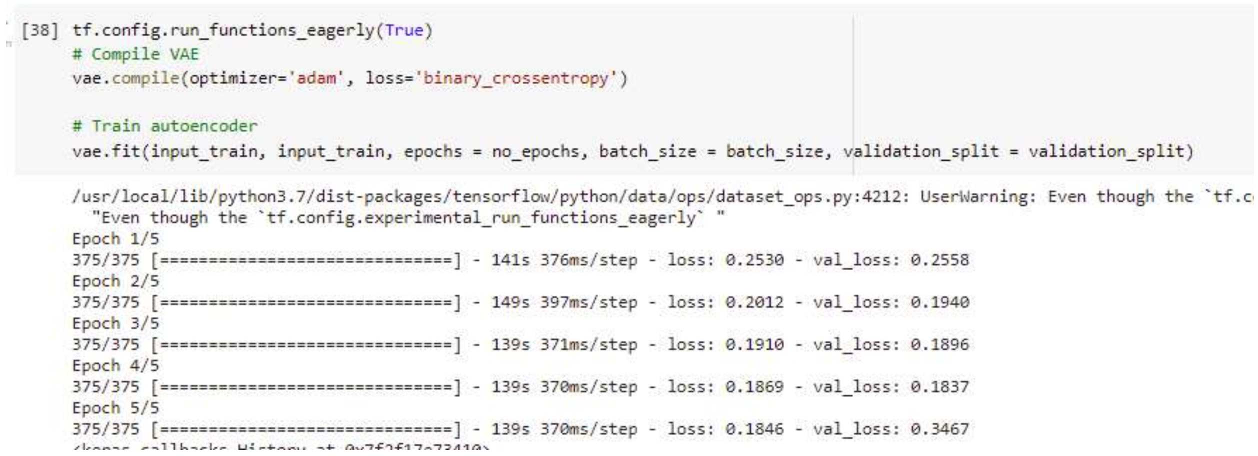
```
[37] # Instantiate VAE
     vae_outputs = decoder(encoder(i)[2])
     vae         = Model(i, vae_outputs, name='vae_enhanced')
     vae.summary()
```

```
Model: "vae_enhanced"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_input (InputLayer)   [(None, 28, 28, 1)]       0
_________________________________________________________________
encoder_enhanced (Functional [(None, 2), (None, 2), (N 7207608
_________________________________________________________________
decoder_enhanced (Functional (None, 28, 28, 1)         15992321
=================================================================
Total params: 23,199,929
Trainable params: 23,091,857
Non-trainable params: 108,072
_________________________________________________________________
```

8. 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.

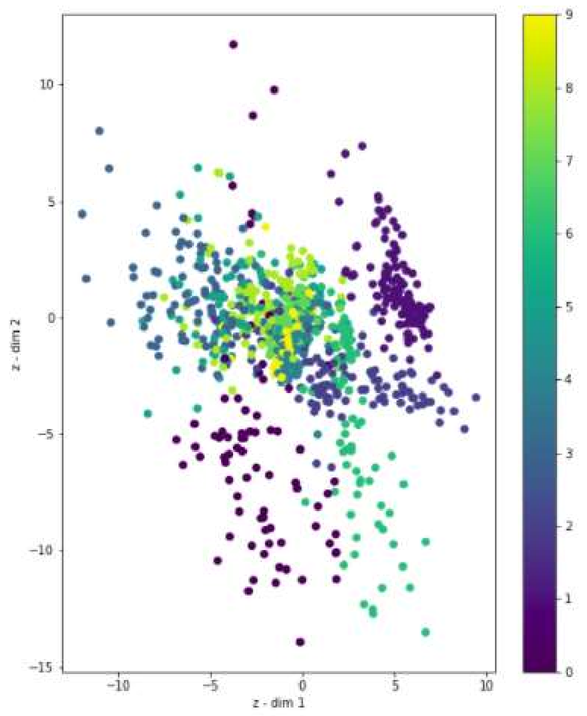


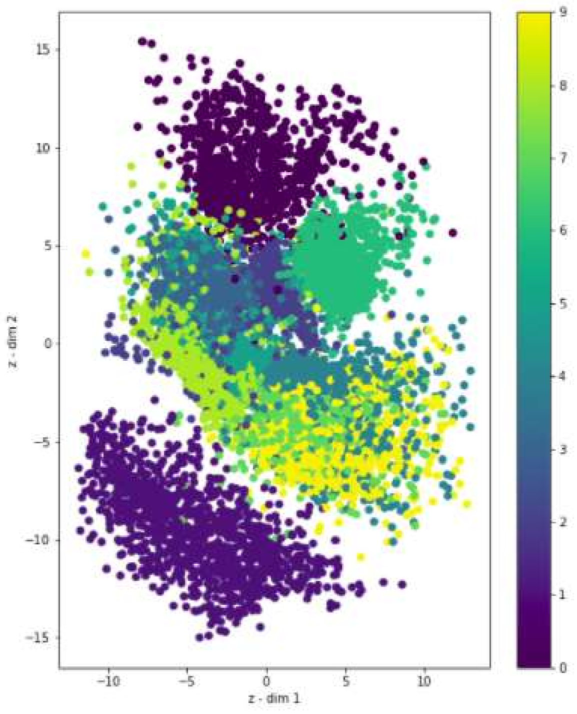
```
[38] tf.config.run_functions_eagerly(True)
     # Compile VAE
     vae.compile(optimizer='adam', loss='binary_crossentropy')

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

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/data/ops/dataset_ops.py:4212: UserWarning: Even though the `tf.c
  "Even though the `tf.config.experimental_run_functions_eagerly` "
Epoch 1/5
375/375 [==============================] - 141s 376ms/step - loss: 0.2530 - val_loss: 0.2558
Epoch 2/5
375/375 [==============================] - 149s 397ms/step - loss: 0.2012 - val_loss: 0.1940
Epoch 3/5
375/375 [==============================] - 139s 371ms/step - loss: 0.1910 - val_loss: 0.1896
Epoch 4/5
375/375 [==============================] - 139s 370ms/step - loss: 0.1869 - val_loss: 0.1837
Epoch 5/5
375/375 [==============================] - 139s 370ms/step - loss: 0.1846 - val_loss: 0.3467
```

9. Visualizing VAE results and comparative remarks

1. Visualizing inputs mapped onto latent space

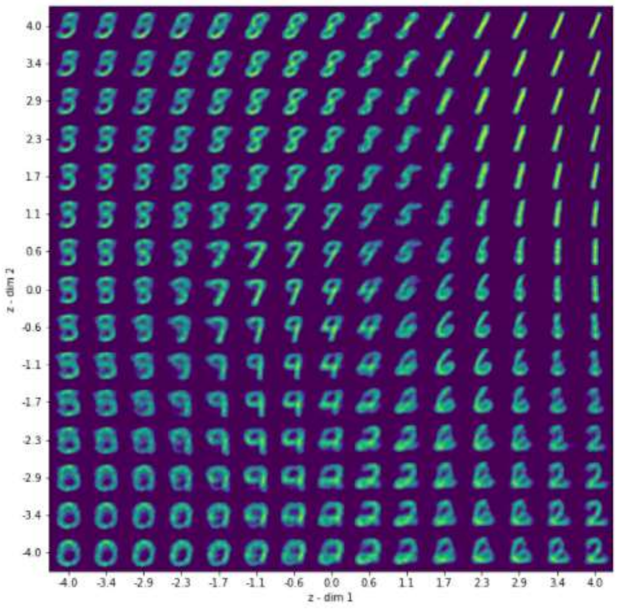




Looking at the output from both the models, we see that enhanced model produces better result than the existing use case.

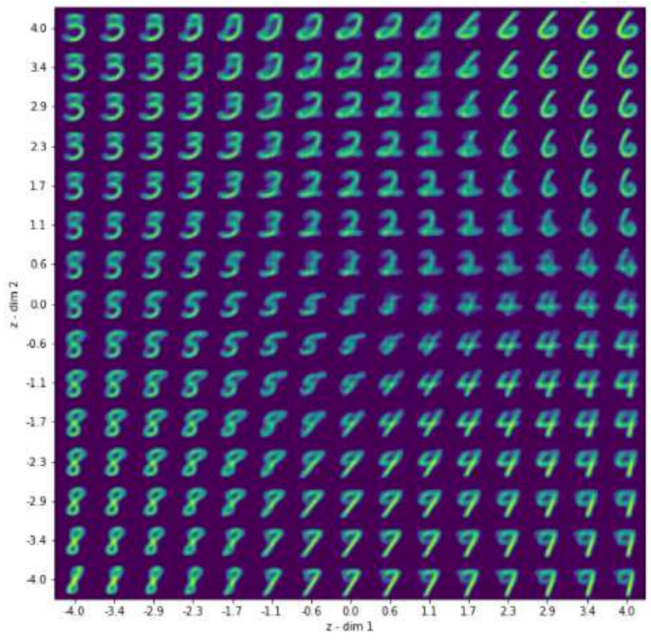
2. Visualizing samples from the latent space

a. Output from the provided use case



We see that especially in the right corners, we see the issue with completeness, which yield outputs that do not make sense. Some issues with continuity are visible wherever the samples are *blurred*.

b. Output after enhancing the VAE –
Compared to the samples in the latent space we observe less noise as compared to the existing use case.



Final Observations:

We clearly see that the enhanced VAE is better than the one already provided in the use case. There are few changes I did to enhance the existing model. They are listed below-

1. **Additional of more Conv2D and Conv2DTranspose layers:** While experimenting with generative models, results clearly show that the deep convolutional architectures for generative models may produce better results with VAEs as well.
2. **Less Number of epochs are required to train the model with more deep layers:** The use case model took 50 epochs whereas enhanced model took just 5 epochs yet produced better results than the use case model.
3. **Visualization of Samples in the latent space has lesser noise than the existing use case.**
4. **Enhanced number of trained parameters:** The enhanced model trained `Total params: 15,992,321` whereas existing use case trained only `Total params: 26,345.`

Video Link

- https://youtu.be/JF1DMdOsSMw

Conclusion

1. Lessons Learnt

- I learnt how to create a variational autoencoder with Keras.
- I understood what VAE’s are and why they are different from regular autoencoders.

2. Challenges Faced

- To increase the number of Conv2D layers, I had to remove strides from the last layer so that encoder and decoder sizes become equal. This solution took a little bit of my time and troubleshooting.
- The right number of Conv2D layers and size selection was a challenge. I could decide on the right number after training the model multiple times.