assignment12

March 5, 2022

[1]:
```
import keras
from keras.layers import Conv2D, Conv2DTranspose, Input, Flatten, Dense,␣
 ↪Lambda, Reshape
from keras.models import Model
from keras.datasets import mnist
from keras import backend as K
import numpy as np
import matplotlib.pyplot as plt
```

```
Using TensorFlow backend.
```

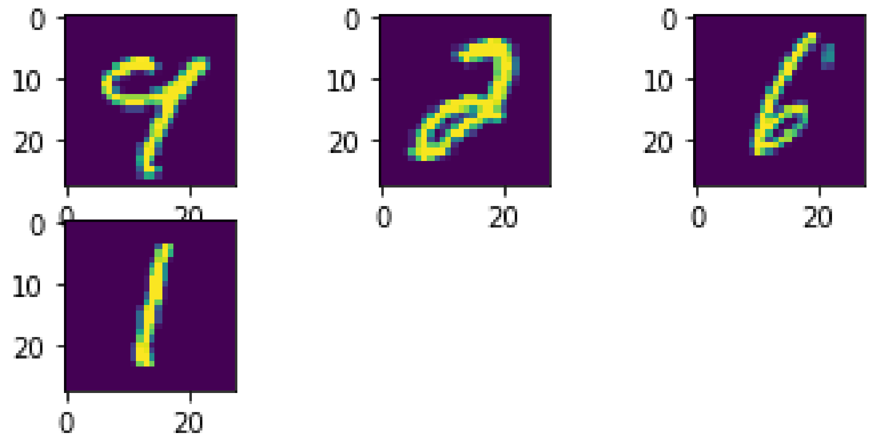
[2]:
```
# Load MNIST
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

[3]:
```
# Normalize and Reshape
# Normalize

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train = x_train / 255
x_test = x_test / 255
```

[4]:
```
# Reshape

img_width = x_train.shape[1]
img_height = x_train.shape[2]
num_channels = 1 # Grey scale data
x_train = x_train.reshape(x_train.shape[0], img_height, img_width, num_channels)
x_test = x_test.reshape(x_test.shape[0], img_height, img_width, num_channels)
input_shape = (img_height, img_width, num_channels)
```

[5]:
```
#View a few images
plt.figure(1)
plt.subplot(331)
plt.imshow(x_train[54][:,:,0])

plt.subplot(332)
plt.imshow(x_train[555][:,:,0])
```

```
plt.subplot(333)
plt.imshow(x_train[6789][:,:,0])

plt.subplot(334)
plt.imshow(x_train[42000][:,:,0])
plt.show()
```



[6]:

```
# Build the model
## Encoder
## Define 4 conv2D, flatten and ten dense
latent_dim = 2 # Number of latent dimension parms(latent space is a 2D plane)
input_img = Input(shape=input_shape, name='encoder_input')
x = Conv2D(32, 3, padding='same', activation='relu')(input_img)
x = Conv2D(64, 3, padding='same', activation='relu',strides=(2, 2))(x)
x = Conv2D(64, 3, padding='same', activation='relu')(x)
x = Conv2D(64, 3, padding='same', activation='relu')(x)
conv_shape = K.int_shape(x) # Shape of conv to be provided to decoder
# Flatten
x = Flatten()(x)
x = Dense(32, activation='relu')(x)
```

[7]:

```
# Two outputs, for latent mean and log variance (std. dev.)
# Use these to sample random variables in latent space to which inputs are␣
↪mapped.

z_mu = Dense(latent_dim, name='latent_mu')(x)   # Mean values of encoded input
z_sigma = Dense(latent_dim, name='latent_sigma')(x)  # Std dev. (variance) of␣
↪encoded input
```

[8]:
```
#REPARAMETERIZATION TRICK
# Define sampling function to sample from the distribution
# Reparameterize sample based on the process defined by Gunderson and Huang
# into the shape of: mu + sigma squared x eps
#This is to allow gradient descent to allow for gradient estimation accurately.
def sample_z(args):
  z_mu, z_sigma = args
  eps = K.random_normal(shape=(K.shape(z_mu)[0], K.int_shape(z_mu)[1]))
  return z_mu + K.exp(z_sigma / 2) * eps
```

[9]:
```
# sample vector from the latent distribution
# z is the lambda custom layer we are adding for gradient descent calculations
# using mu and variance (sigma)
z = Lambda(sample_z, output_shape=(latent_dim, ), name='z')([z_mu, z_sigma])

#Z (lambda layer) will be the last layer in the encoder.
# Define and summarize encoder model.
encoder = Model(input_img, [z_mu, z_sigma, z], name='encoder')
print(encoder.summary())
```

```
Model: "encoder"
________________________________________________________________________________
__________________
Layer (type)                    Output Shape          Param #     Connected to
================================================================================
==================
encoder_input (InputLayer)      (None, 28, 28, 1)     0
________________________________________________________________________________
__________________
conv2d_1 (Conv2D)               (None, 28, 28, 32)    320
encoder_input[0][0]
________________________________________________________________________________
__________________
conv2d_2 (Conv2D)               (None, 14, 14, 64)    18496       conv2d_1[0][0]
________________________________________________________________________________
__________________
conv2d_3 (Conv2D)               (None, 14, 14, 64)    36928       conv2d_2[0][0]
________________________________________________________________________________
__________________
conv2d_4 (Conv2D)               (None, 14, 14, 64)    36928       conv2d_3[0][0]
________________________________________________________________________________
__________________
flatten_1 (Flatten)             (None, 12544)         0           conv2d_4[0][0]
________________________________________________________________________________
__________________
dense_1 (Dense)                 (None, 32)            401440      flatten_1[0][0]
________________________________________________________________________________
__________________
```

```
latent_mu (Dense)               (None, 2)            66          dense_1[0][0]
--------------------------------------------------------------------------------
------------------
latent_sigma (Dense)            (None, 2)            66          dense_1[0][0]
--------------------------------------------------------------------------------
------------------
z (Lambda)                      (None, 2)            0           latent_mu[0][0]
latent_sigma[0][0]
================================================================================
==================
Total params: 494,244
Trainable params: 494,244
Non-trainable params: 0
--------------------------------------------------------------------------------
------------------
None
```

[10]:

```
# decoder takes the latent vector as input
decoder_input = Input(shape=(latent_dim,),name ='decoder_input')
# Need to start with a shape that can be remapped to original image shape
# add dense layer with dimensions that can be resaped to desired output shape
x = Dense(conv_shape[1]*conv_shape[2]*conv_shape[3],␣
↪activation='relu')(decoder_input)
# reshape to the shape of last conv. layer in the encoder, so we can
x = Reshape((conv_shape[1], conv_shape[2], conv_shape[3]))(x)
# upscale (conv2D transpose) back to original shape
# use Conv2DTranspose to reverse the conv layers defined in the encoder
x = Conv2DTranspose(32, 3, padding='same', activation='relu',strides=(2, 2))(x)
#Can add more conv2DTranspose layers, if desired.
#Using sigmoid activation
x = Conv2DTranspose(num_channels, 3, padding='same', activation='sigmoid',␣
↪name='decoder_output')(x)

# Define and summarize decoder model
decoder = Model(decoder_input, x, name='decoder')
decoder.summary()

# apply the decoder to the latent sample
z_decoded = decoder(z)
```

```
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
decoder_input (InputLayer)   (None, 2)                 0
_________________________________________________________________
dense_2 (Dense)              (None, 12544)             37632
_________________________________________________________________
```

```
reshape_1 (Reshape)          (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 28, 28, 32)        18464
_________________________________________________________________
decoder_output (Conv2DTransp (None, 28, 28, 1)         289
=================================================================
Total params: 56,385
Trainable params: 56,385
Non-trainable params: 0
_________________________________________________________________
```

[11]:
```python
#Define custom loss
#VAE is trained using two loss functions reconstruction loss and KL divergence
#Let us add a class to define a custom layer with loss
class CustomLayer(keras.layers.Layer):

    def vae_loss(self, x, z_decoded):
        x = K.flatten(x)
        z_decoded = K.flatten(z_decoded)

        # Reconstruction loss (as we used sigmoid activation we can use␣
↪binarycrossentropy)
        recon_loss = keras.metrics.binary_crossentropy(x, z_decoded)

        # KL divergence
        kl_loss = -5e-4 * K.mean(1 + z_sigma - K.square(z_mu) - K.exp(z_sigma),␣
↪axis=-1)
        return K.mean(recon_loss + kl_loss)

    # add custom loss to the class
    def call(self, inputs):
        x = inputs[0]
        z_decoded = inputs[1]
        loss = self.vae_loss(x, z_decoded)
        self.add_loss(loss, inputs=inputs)
        return x
```

[12]:
```python
# apply the custom loss to the input images and the decoded latent distribution␣
↪sample
y = CustomLayer()([input_img, z_decoded])
# y is basically the original image after encoding input img to mu, sigma, z
# and decoding sampled z values.
#This will be used as output for vae
```

[13]:
```python
vae = Model(input_img, y, name='vae')

# Compile VAE
```

```
vae.compile(optimizer='adam', loss=None)
vae.summary()
```

```
Model: "vae"
________________________________________________________________________________
__________________
Layer (type)                    Output Shape         Param #     Connected to
================================================================================
==================
encoder_input (InputLayer)      (None, 28, 28, 1)    0

________________________________________________________________________________
__________________
conv2d_1 (Conv2D)               (None, 28, 28, 32)   320
encoder_input[0][0]

________________________________________________________________________________
__________________
conv2d_2 (Conv2D)               (None, 14, 14, 64)   18496       conv2d_1[0][0]

________________________________________________________________________________
__________________
conv2d_3 (Conv2D)               (None, 14, 14, 64)   36928       conv2d_2[0][0]

________________________________________________________________________________
__________________
conv2d_4 (Conv2D)               (None, 14, 14, 64)   36928       conv2d_3[0][0]

________________________________________________________________________________
__________________
flatten_1 (Flatten)             (None, 12544)        0           conv2d_4[0][0]

________________________________________________________________________________
__________________
dense_1 (Dense)                 (None, 32)           401440      flatten_1[0][0]

________________________________________________________________________________
__________________
latent_mu (Dense)               (None, 2)            66          dense_1[0][0]

________________________________________________________________________________
__________________
latent_sigma (Dense)            (None, 2)            66          dense_1[0][0]

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

________________________________________________________________________________
__________________
decoder (Model)                 (None, 28, 28, 1)    56385       z[0][0]

________________________________________________________________________________
__________________
custom_layer_1 (CustomLayer)    [(None, 28, 28, 1),  0
encoder_input[0][0]
                                                                 decoder[1][0]
================================================================================
```

```
==================
Total params: 550,629
Trainable params: 550,629
Non-trainable params: 0
________________________________________________________________________________
__________________
C:\Users\saman\.conda\envs\dsc650\lib\site-
packages\keras\engine\training_utils.py:819: UserWarning: Output custom_layer_1
missing from loss dictionary. We assume this was done on purpose. The fit and
evaluate APIs will not be expecting any data to be passed to custom_layer_1.
  'be expecting any data to be passed to {0}.'.format(name))
```

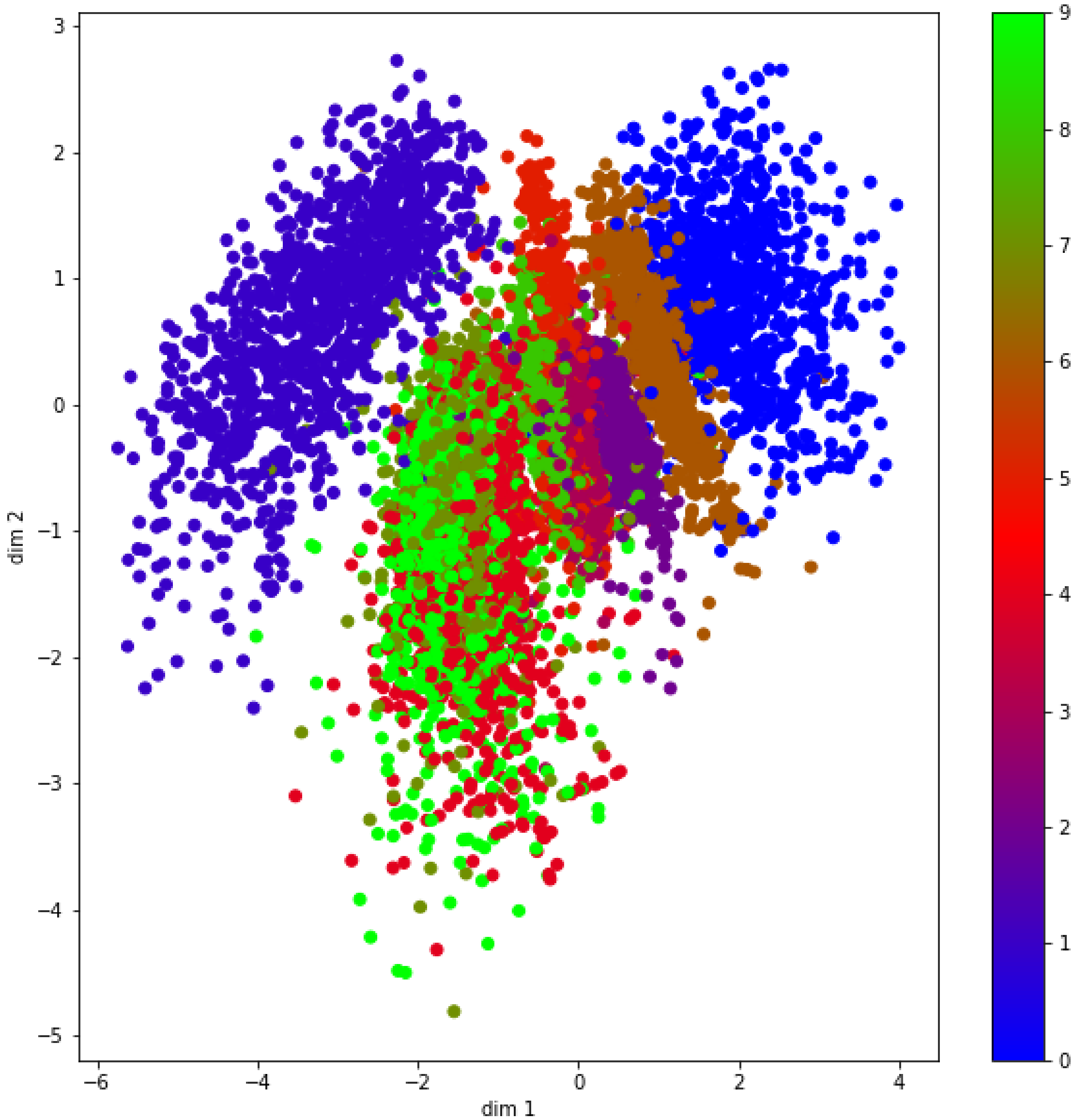
[14]:

```python
# Train autoencoder
vae.fit(x_train, None, epochs = 10, batch_size = 32, validation_split = 0.2)
```

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/10
48000/48000 [==============================] - 109s 2ms/step - loss: 0.2315 -
val_loss: 0.2085
Epoch 2/10
48000/48000 [==============================] - 118s 2ms/step - loss: 0.2021 -
val_loss: 0.1974
Epoch 3/10
48000/48000 [==============================] - 105s 2ms/step - loss: 0.1953 -
val_loss: 0.1934
Epoch 4/10
48000/48000 [==============================] - 112s 2ms/step - loss: 0.1919 -
val_loss: 0.1907
Epoch 5/10
48000/48000 [==============================] - 110s 2ms/step - loss: 0.1895 -
val_loss: 0.1885
Epoch 6/10
48000/48000 [==============================] - 117s 2ms/step - loss: 0.1880 -
val_loss: 0.1876
Epoch 7/10
48000/48000 [==============================] - 113s 2ms/step - loss: 0.1866 -
val_loss: 0.1866
Epoch 8/10
48000/48000 [==============================] - 108s 2ms/step - loss: 0.1856 -
val_loss: 0.1861
Epoch 9/10
48000/48000 [==============================] - 113s 2ms/step - loss: 0.1847 -
val_loss: 0.1852
Epoch 10/10
48000/48000 [==============================] - 120s 3ms/step - loss: 0.1839 -
val_loss: 0.1859
```

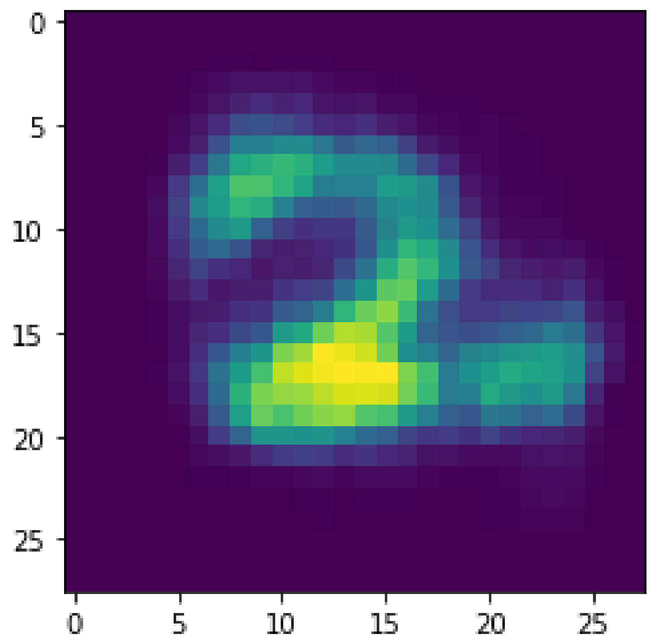
[14]: <keras.callbacks.callbacks.History at 0x1f556b73f98>

[15]:
```
# =================
# Visualize results
# =================
#Visualize inputs mapped to the Latent space
#Remember that we have encoded inputs to latent space dimension = 2.
#Extract z_mu --> first parameter in the result of encoder prediction␣
↪representing mean

mu, _, _ = encoder.predict(x_test)
#Plot dim1 and dim2 for mu
plt.figure(figsize=(10, 10))
plt.scatter(mu[:, 0], mu[:, 1], c=y_test, cmap='brg')
plt.xlabel('dim 1')
plt.ylabel('dim 2')
plt.colorbar()
plt.show()
```



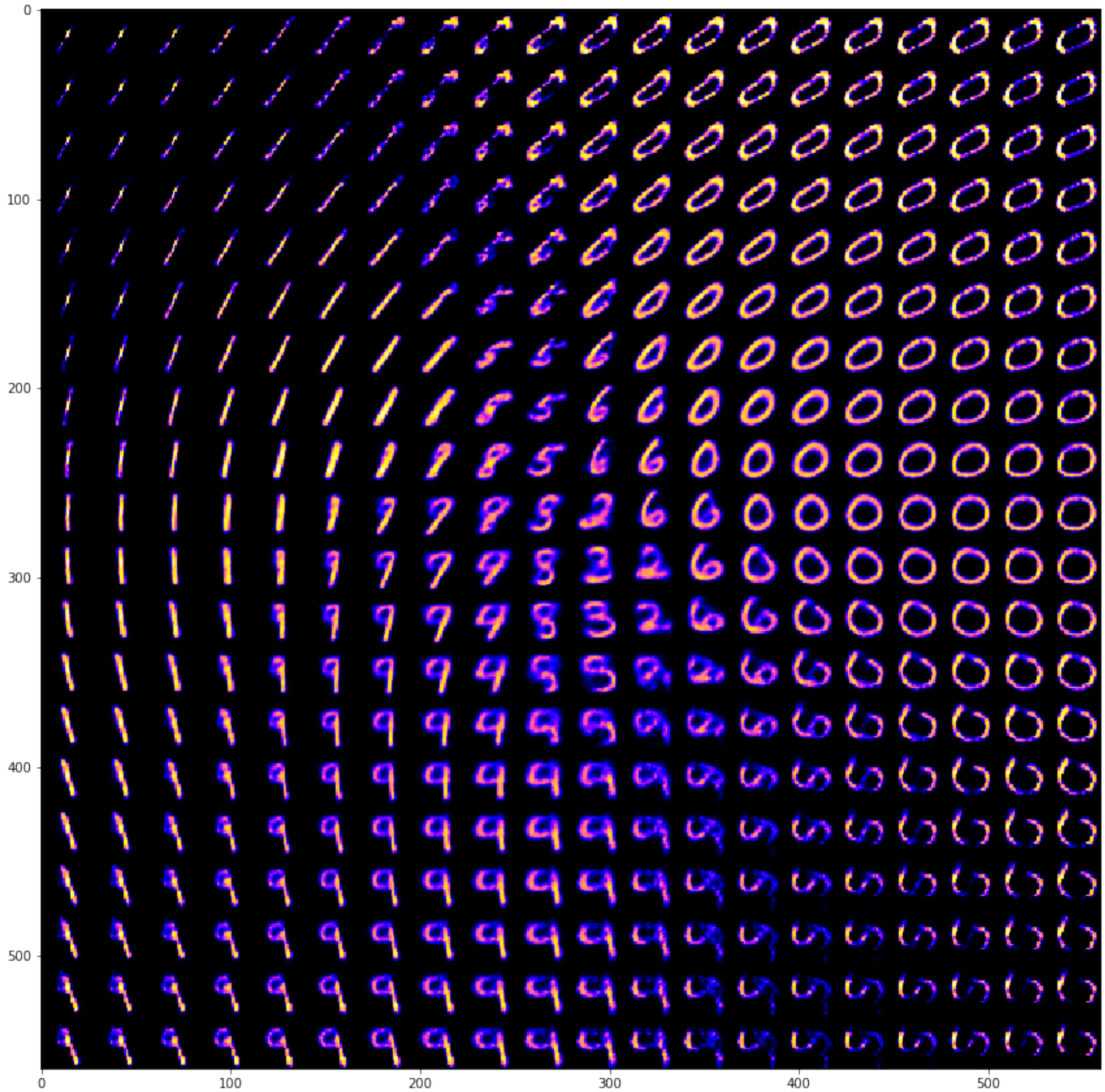
[16]:
```
# Visualize images
#Single decoded image with random input latent vector (of size 1x2)
#Latent space range is about -5 to 5 so pick random values within this range
#Try starting with -1, 1 and slowly go up to -1.5,1.5 and see how it morphs␣
↪from
#one image to the other.
sample_vector = np.array([[1,-1]])
decoded_example = decoder.predict(sample_vector)
decoded_example_reshaped = decoded_example.reshape(img_width, img_height)
plt.imshow(decoded_example_reshaped)
```

[16]: <matplotlib.image.AxesImage at 0x1f55f665da0>



[17]:
```
#Let us automate this process by generating multiple images and plotting
#Use decoder to generate images by tweaking latent variables from the latent␣
↪space
#Create a grid of defined size with zeros.
#Take sample from some defined linear space. In this example range [-4, 4]
#Feed it to the decoder and update zeros in the figure with output.
import os
from pathlib import Path

n = 20  # generate 15x15 digits
figure = np.zeros((img_width * n, img_height * n, num_channels))
#Create a Grid of latent variables, to be provided as inputs to decoder.predict
#Creating vectors within range -5 to 5 as that seems to be the range in latent␣
↪space
grid_x = np.linspace(-5, 5, n)
grid_y = np.linspace(-5, 5, n)[::-1]

# decoder for each square in the grid
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=(15, 15))
#Reshape for visualization
fig_shape = np.shape(figure)
figure = figure.reshape((fig_shape[0], fig_shape[1]))

plt.imshow(figure, cmap='gnuplot2')

current_dir = Path(os.getcwd()).absolute()
results_dir = current_dir.joinpath('results')
results_dir.mkdir(parents=True, exist_ok=True)
vae_dir =results_dir.joinpath('vae')
vae_dir.mkdir(parents=True, exist_ok=True)

plt.savefig(f'{vae_dir}\\result.png')

plt.show()
```


0
100
200
300
400
500
0
100
200
300
400
500

[ ]: