

Image_generation

May 31, 2021

1 Week 12

1.1 Images Generation

1.2 Gloria Moore

```
[1]: import keras
import tensorflow as tf
keras.__version__
```

```
[1]: '2.4.3'
```

```
[2]: from keras import backend as K
K.clear_session()
```

```
[3]: import numpy as np
from keras import layers
import os
import matplotlib as plt
import tensorflow.compat.v1.keras.backend as k
from pathlib import Path
from scipy.stats import norm
from keras.datasets import mnist
from tensorflow.keras.models import Model
```

```
[4]: #from Robert Code

tf.compat.v1.disable_eager_execution()

tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
```

```
[5]: #From the book

img_shape = (28, 28, 1)
batch_size = 16
latent_dim = 2 # Dimensionality of the latent space: a plane

input_img = keras.Input(shape=img_shape)
```

```

x = layers.Conv2D(32, 3,
                  padding='same', activation='relu')(input_img)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu',
                  strides=(2, 2))(x)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
shape_before_flattening = K.int_shape(x)

x = layers.Flatten()(x)
x = layers.Dense(32, activation='relu')(x)

z_mean = layers.Dense(latent_dim)(x)
z_log_var = layers.Dense(latent_dim)(x)

```

```

[6]: def sampling(args):
      z_mean, z_log_var = args
      epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
                                mean=0., stddev=1.)
      return z_mean + K.exp(z_log_var) * epsilon

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

```

```

[7]: # This is the input where we will feed `z`.
      decoder_input = layers.Input(K.int_shape(z)[1:])

      # Upsample to the correct number of units
      x = layers.Dense(np.prod(shape_before_flattening[1:]),
                       activation='relu')(decoder_input)

      # Reshape into an image of the same shape as before our last `Flatten` layer
      x = layers.Reshape(shape_before_flattening[1:])(x)

      # We then apply then reverse operation to the initial
      # stack of convolution layers: a `Conv2DTranspose` layers
      # with corresponding parameters.
      x = layers.Conv2DTranspose(32, 3,
                                padding='same', activation='relu',
                                strides=(2, 2))(x)

      x = layers.Conv2D(1, 3,
                        padding='same', activation='sigmoid')(x)
      # We end up with a feature map of the same size as the original input.

      # This is our decoder model.

```

```

decoder = Model(decoder_input, x)

# We then apply it to `z` to recover the decoded `z`.
z_decoded = decoder(z)

```

```

[8]: class CustomVariationalLayer(keras.layers.Layer):

    def vae_loss(self, x, z_decoded):
        x = K.flatten(x)
        z_decoded = K.flatten(z_decoded)
        xent_loss = keras.metrics.binary_crossentropy(x, z_decoded)
        kl_loss = -5e-4 * K.mean(
            1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)
        return K.mean(xent_loss + kl_loss)

    def call(self, inputs):
        x = inputs[0]
        z_decoded = inputs[1]
        loss = self.vae_loss(x, z_decoded)
        self.add_loss(loss, inputs=inputs)
        # We don't use this output.
        return x

# We call our custom layer on the input and the decoded output,
# to obtain the final model output.
y = CustomVariationalLayer()([input_img, z_decoded])

```

```

[9]: vae = Model(input_img, y)
vae.compile(optimizer='rmsprop', loss=None)
vae.summary()

# Train the VAE on MNIST digits
(x_train, _), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_train = x_train.reshape(x_train.shape + (1,))
x_test = x_test.astype('float32') / 255.
x_test = x_test.reshape(x_test.shape + (1,))

vae.fit(x=x_train, y=None,
        shuffle=True,
        epochs=10,
        batch_size=batch_size,
        validation_data=(x_test, None))

```

Model: "model_1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 28, 28, 1)]	0	
conv2d (Conv2D)	(None, 28, 28, 32)	320	input_1[0][0]
conv2d_1 (Conv2D)	(None, 14, 14, 64)	18496	conv2d[0][0]
conv2d_2 (Conv2D)	(None, 14, 14, 64)	36928	conv2d_1[0][0]
conv2d_3 (Conv2D)	(None, 14, 14, 64)	36928	conv2d_2[0][0]
flatten (Flatten)	(None, 12544)	0	conv2d_3[0][0]
dense (Dense)	(None, 32)	401440	flatten[0][0]
dense_1 (Dense)	(None, 2)	66	dense[0][0]
dense_2 (Dense)	(None, 2)	66	dense[0][0]
lambda (Lambda)	(None, 2)	0	dense_1[0][0] dense_2[0][0]
model (Functional)	(None, 28, 28, 1)	56385	lambda[0][0]
custom_variational_layer (Custo	(None, 28, 28, 1)	0	input_1[0][0] model[0][0]

Total params: 550,629
 Trainable params: 550,629
 Non-trainable params: 0

Train on 60000 samples, validate on 10000 samples

```

Epoch 1/10
59968/60000 [=====>.] - ETA: 0s - loss: 0.2095

/opt/conda/lib/python3.8/site-
packages/tensorflow/python/keras/engine/training.py:2325: UserWarning:
`Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  warnings.warn("`Model.state_updates` will be removed in a future version. '
60000/60000 [=====] - 86s 1ms/sample - loss: 0.2095 -
val_loss: 0.1950
Epoch 2/10
60000/60000 [=====] - 83s 1ms/sample - loss: 0.1917 -
val_loss: 0.1880
Epoch 3/10
60000/60000 [=====] - 83s 1ms/sample - loss: 0.1876 -
val_loss: 0.1862
Epoch 4/10
60000/60000 [=====] - 83s 1ms/sample - loss: 0.1853 -
val_loss: 0.1850
Epoch 5/10
60000/60000 [=====] - 83s 1ms/sample - loss: 0.1837 -
val_loss: 0.1829
Epoch 6/10
60000/60000 [=====] - 83s 1ms/sample - loss: 0.1826 -
val_loss: 0.1831
Epoch 7/10
60000/60000 [=====] - 83s 1ms/sample - loss: 0.1818 -
val_loss: 0.1819
Epoch 8/10
60000/60000 [=====] - 84s 1ms/sample - loss: 0.1811 -
val_loss: 0.1815
Epoch 9/10
60000/60000 [=====] - 85s 1ms/sample - loss: 0.1804 -
val_loss: 0.1815
Epoch 10/10
60000/60000 [=====] - 83s 1ms/sample - loss: 0.1799 -
val_loss: 0.1806

```

```
[9]: <tensorflow.python.keras.callbacks.History at 0x7fc526186490>
```

```
[11]: current_dir = Path(os.getcwd()).absolute()
      vae_dir = current_dir.joinpath('results').joinpath('vae')
```

```
[12]: import matplotlib.pyplot as plt

      # Display a 2D manifold of the digits
```

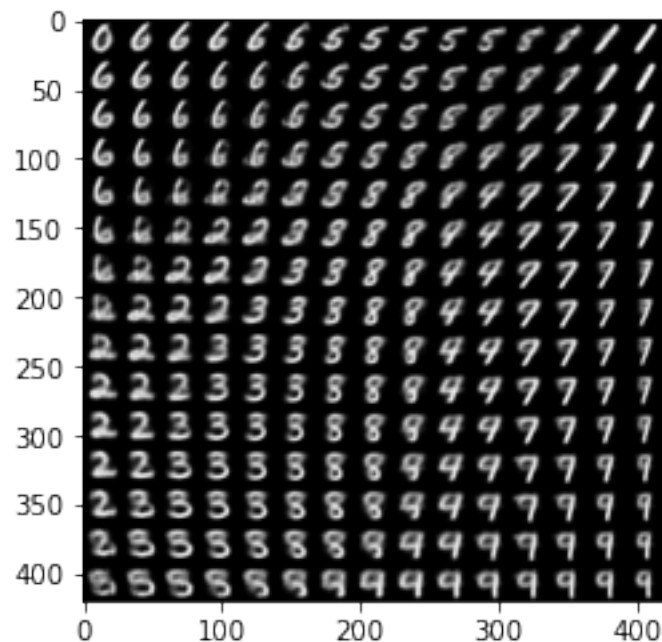
```

n = 15 # figure with 15x15 digits
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
# Linearly spaced coordinates on the unit square were transformed
# through the inverse CDF (ppf) of the Gaussian
# to produce values of the latent variables z,
# since the prior of the latent space is Gaussian
grid_x = norm.ppf(np.linspace(0.05, 0.95, n))
grid_y = norm.ppf(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]])
        z_sample = np.tile(z_sample, batch_size).reshape(batch_size, 2)
        x_decoded = decoder.predict(z_sample, batch_size=batch_size)
        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, cmap='Greys_r')
plt.savefig(vae_dir.joinpath('15x15_Grid_VAE_encod_MNIST_nums.png'),
            bbox_inches='tight')
plt.show()

```



```
[ ]: plt.savefig(vae_dir.joinpath('15x15_Grid_VAE_encod_MNIST_nums.png'),  
↳ bbox_inches='tight')
```

```
[ ]:
```