1000-719bMSB Modeling of Complex Biological Systems

Deep Neural Network: Unsupervised Learning

Variational Autoencoder on MNIST

VAEs are directed probabilistic graphical models (DPGM) whose posterior is approximated by a neural network, forming an autoencoder-like architecture. Unlike discriminative modeling that aims to learn a predictor given observation, generative modeling tries to learn how the data is generated, and to reflect the underlying causal relations. [...] Variational autoencoder models make strong assumptions concerning the distribution of latent variables. They use a variational approach for latent representation learning, which results in an additional loss component and a specific estimator for the training algorithm called the Stochastic Gradient Variational Bayes (SGVB) estimator. Wikipedia

We build a VAE and train it on the MNIST dataset.

In [2]:
```
#%tensorflow_version 1.x
import tensorflow as tf
print(tf.__version__)
tf.compat.v1.disable_eager_execution()
import keras
from keras import layers
from keras import backend as K
from keras.models import Model
from keras.datasets import mnist
import matplotlib.pyplot as plt
import numpy as np


img_shape = (28, 28, 1)
batch_size = 16
latent_dim = 2
```
```
2.7.0
```

In [3]:
```
#Define encoder network
#Note that we are using keras functional API
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)
#x = layers.Dense(2, activation='relu')(x)

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

In [4]:
```
#Sampling from the distributions to obtain latent space
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])

encoder = Model(input_img, z)
```

In [5]:
```
#Define decoder network
decoder_input = layers.Input(K.int_shape(z)[1:])
x = layers.Dense(np.prod(shape_before_flattening[1:]), activation='relu')(decoder_input)
x = layers.Reshape(shape_before_flattening[1:])(x)
x = layers.Conv2DTranspose(32, 3,padding='same', activation='relu', strides=(2, 2))(x)
x = layers.Conv2D(1, 3,padding='same', activation='sigmoid')(x)

decoder = Model(decoder_input, x)
z_decoded = decoder(z)
```

In [6]:
```
def vae_loss(input_img, z_decoded):
    input_img = K.flatten(input_img)
    z_decoded = K.flatten(z_decoded)
    xent_loss = keras.metrics.binary_crossentropy(input_img, 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)
```

In [7]:
```
vae = Model(input_img, z_decoded)
vae.compile(optimizer='adam', loss=vae_loss)
```

In [8]:
```
vae.summary()
decoder.summary()
```
```
Model: "model_2"
__________________________________________________________________________________________________
 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_1 (Functional)           (None, 28, 28, 1)    56385       ['lambda[0][0]']

==================================================================================================
Total params: 550,629
Trainable params: 550,629
Non-trainable params: 0
__________________________________________________________________________________________________
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 2)]               0

 dense_3 (Dense)             (None, 12544)             37632

 reshape (Reshape)           (None, 14, 14, 64)        0

 conv2d_transpose (Conv2DTra  (None, 28, 28, 32)       18464
 nspose)

 conv2d_4 (Conv2D)           (None, 28, 28, 1)         289

=================================================================
Total params: 56,385
Trainable params: 56,385
Non-trainable params: 0
_________________________________________________________________
```

In [9]:
```
#load the data and split into train + test sets
(x_train, _), (x_test, y_test) = mnist.load_data()
```

In [10]:
```
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,))
```

We are ready to train VAE on this dataset

In [11]:
```
vae.fit(x=x_train,y=x_train, shuffle=True, epochs=10, batch_size=batch_size)
```
```
Train on 60000 samples
Epoch 1/10
60000/60000 [==============================] - 182s 3ms/sample - loss: 0.2272
Epoch 2/10
60000/60000 [==============================] - 174s 3ms/sample - loss: 0.2013
Epoch 3/10
60000/60000 [==============================] - 177s 3ms/sample - loss: 0.1947
Epoch 4/10
60000/60000 [==============================] - 197s 3ms/sample - loss: 0.1909
Epoch 5/10
60000/60000 [==============================] - 208s 3ms/sample - loss: 0.1885
Epoch 6/10
60000/60000 [==============================] - 183s 3ms/sample - loss: 0.1865
Epoch 7/10
60000/60000 [==============================] - 185s 3ms/sample - loss: 0.1851
Epoch 8/10
60000/60000 [==============================] - 182s 3ms/sample - loss: 0.1840
Epoch 9/10
60000/60000 [==============================] - 178s 3ms/sample - loss: 0.1831
Epoch 10/10
60000/60000 [==============================] - 174s 3ms/sample - loss: 0.1824
```

Out[11]:
```
<keras.callbacks.History at 0x1cde7140610>
```

In [12]:
```
# to save the trained VAE on Google Drive and load them later. Not necessary
from google.colab import drive
drive.mount('/content/drive')
```
```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-12-de0121779945> in <module>
      1 # to save the trained VAE on Google Drive and load them later. Not necessary
----> 2 from google.colab import drive
      3 drive.mount('/content/drive')

ModuleNotFoundError: No module named 'google.colab'
```

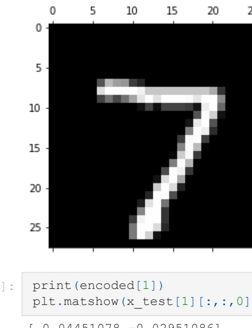
In [ ]:
```
vae.save_weights('drive/My Drive/Colab Notebooks/vae_weights.h5')
vae.load_weights('drive/My Drive/Colab Notebooks/vae_weights.h5')
```

In [13]:
```
# x_test is data
# encoded: means we are mapping our data point (x_test) into latent space; give us a point in latent space
encoded = encoder.predict(x_test)
```
```
C:\Users\zosia\anaconda3\lib\site-packages\keras\engine\training_v1.py:2079: 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.
  updates=self.state_updates,
```

In [14]:
```
print(encoded)
```
```
[[ 2.36461    -1.4998004 ]
 [ 0.04451078 -0.02951086]
 [-1.3342675  -3.2245126 ]
 ...
 [ 1.2987785  -0.90321654]
 [-0.9657645  -0.82044506]
 [-0.5781946   0.19804764]]
```

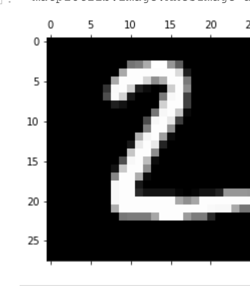
In [15]:
```
print(encoded[0])
plt.matshow(x_test[0][:,:,0], cmap='gray')
```
```
[ 2.36461   -1.4998004]
```

Out[15]:
```
<matplotlib.image.AxesImage at 0x1cde887b910>
```



In [16]:
```
print(encoded[1])
plt.matshow(x_test[1][:,:,0], cmap='gray')
```
```
[ 0.04451078 -0.02951086]
```

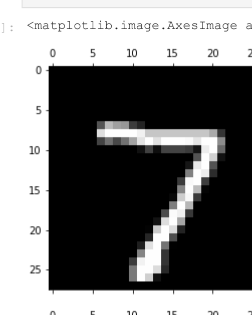
Out[16]:
```
<matplotlib.image.AxesImage at 0x1cde894da90>
```

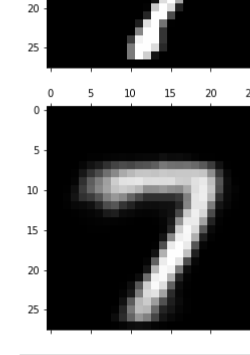


In [17]:
```
# decoded: reconstructing
decoded = decoder.predict(encoded)

#decoded_some_random = decoder.predict()
```

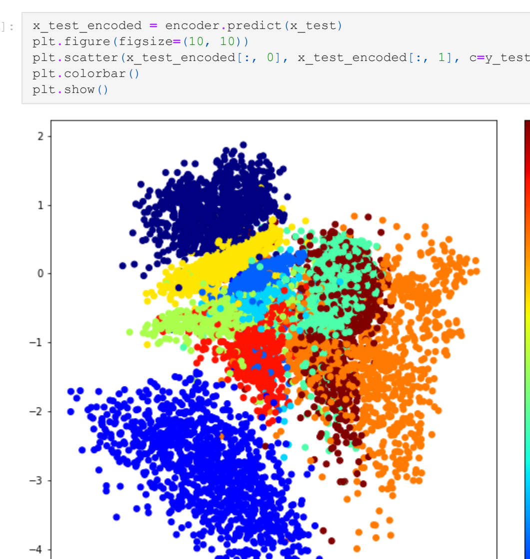
In [18]:
```
plt.matshow(x_test[0][:,:,0], cmap='gray')
plt.matshow(decoded[0][:,:,0], cmap='gray')
```

Out[18]:
```
<matplotlib.image.AxesImage at 0x1cde9b64400>
```





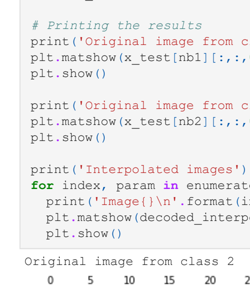
In [19]:
```
x_test_encoded = encoder.predict(x_test)
plt.figure(figsize=(10, 10))
plt.scatter(x_test_encoded[:, 0], x_test_encoded[:, 1], c=y_test, cmap='jet')
plt.colorbar()
plt.show()
```



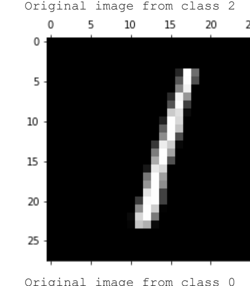
HOMEWORK 1

Interpolate between two latent vectors (i.e., moving in the latent space) and decode and visualize the interpolations. Provide a code (python notebook) to do this interpolation; and provide 3 interpolated images in-between.

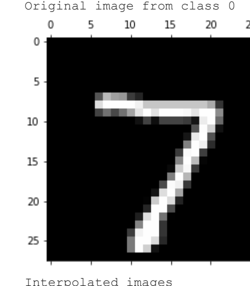
In [20]:
```
def interpolate(vec1, vec2, nb):
    linspace = np.linspace(0, 3, num=nb)
    vectors = []
    for lin in linspace:
      v_new = (1.0 - lin) * vec1 + lin * vec2
      vectors.append(v_new)
    return np.asarray(vectors)

#selecting pictures
nb1 = 2
nb2 = 0

encoded_1 = encoded[nb1]
encoded_2 = encoded[nb2]


# interpolating beetween chosen pictures and decoding them
interpolated_vector = interpolate(encoded_1, encoded_2, 3)
decoded_interpolation = decoder.predict(interpolated_vector)

# Printing the results
print('Original image from class ' + str(nb1))
plt.matshow(x_test[nb1][:,:,0], cmap='gray')
plt.show()

print('Original image from class ' + str(nb2))
plt.matshow(x_test[nb2][:,:,0], cmap='gray')
plt.show()

print('Interpolated images')
for index, param in enumerate(decoded_interpolation):
  print('Image{}\n'.format(index+1))
  plt.matshow(decoded_interpolation[index][:,:,0], cmap='gray')
  plt.show()
```
```
Original image from class 2
```



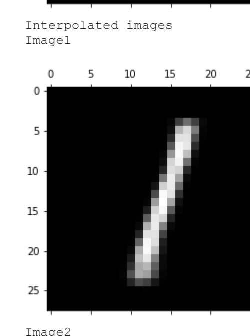
```
Original image from class 0
```



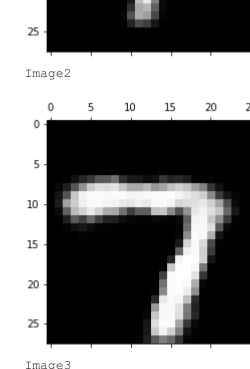
```
Interpolated images
Image1
```



```
Image2
```



```
Image3
```



DeepDream

Read the introduction to DeepDream

In [ ]:
```
from keras.applications import inception_v3
from keras import backend as K
#disable all training-specific options
K.set_learning_phase(0)
```

In [ ]:
```
model = inception_v3.InceptionV3(weights='imagenet',
include_top=False)
```

In [ ]:
```
model.summary()
```

In [ ]:
```
#coefficients for contribution of selected layers
layer_contributions = {
'mixed2': 0.2,
'mixed3': 3.,
'mixed4': 2.,
'mixed5': 1.5,
}
```

In [ ]:
```
#Creates a dictionary that maps layer names to layer instances
layer_dict = dict([(layer.name, layer) for layer in model.layers])
```

In [ ]:
```
layer_dict['mixed2']
```

In [ ]:
```
#define the loss
loss = K.variable(0.)
for layer_name in layer_contributions:
    coeff = layer_contributions[layer_name]
    activation = layer_dict[layer_name].output
    scaling = K.prod(K.cast(K.shape(activation), 'float32'))
    #here the loss is the L2 norm of activations of whole layer
    #before: only chosen filter
    loss = loss + coeff * K.sum(K.square(activation[:, 2: -2, 2: -2, :])) / scaling
```

In [ ]:
```
#define gradient-ascent process

#this was before a noise image, now image of choice
dream = model.input

#gradient of loss wrt input image
grads = K.gradients(loss, dream)[0]
#normalize gradient
grads /= K.maximum(K.mean(K.abs(grads)), 1e-7)

outputs = [loss, grads]
fetch_loss_and_grads = K.function([dream], outputs)

def eval_loss_and_grads(x):
    outs = fetch_loss_and_grads([x])
    loss_value = outs[0]
    grad_values = outs[1]
    return loss_value, grad_values

def gradient_ascent(x, iterations, step, max_loss=None):
    for i in range(iterations):
        loss_value, grad_values = eval_loss_and_grads(x)
        if max_loss is not None and loss_value > max_loss:
            break
        print('...Loss value at', i, ':', loss_value)
        x = x + step * grad_values
    return x
```

In [ ]:
```
#define some auxilliary function
import scipy
from keras.preprocessing import image
import imageio
def resize_img(img, size):
    img = np.copy(img)
    factors = (1,
    float(size[0]) / img.shape[1],
    float(size[1]) / img.shape[2],
    1)
    return scipy.ndimage.zoom(img, factors, order=1)
def save_img(img, fname):
    pil_img = deprocess_image(np.copy(img))
    imageio.imwrite(fname, pil_img)
def preprocess_image(image_path):
    img = image.load_img(image_path, target_size=(400,400))
    img = image.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = inception_v3.preprocess_input(img)
    return img
def deprocess_image(x):
    if K.image_data_format() == 'channels_first':
        x = x.reshape((3, x.shape[2], x.shape[3]))
        x = x.transpose((1, 2, 0))
    else:
        #undoes the preprocessing done by inception_v3.preprocess_input
        x = x.reshape((x.shape[1], x.shape[2], 3))
        x /= 2.
        x += 0.5
        x *= 255.
        x = np.clip(x, 0, 255).astype('uint8')
    return x
```

In [ ]:
```
base_image_path = 'drive/My Drive/Colab Notebooks/hill_wiki.jpg'
img = preprocess_image(base_image_path)
```

In [ ]:
```
step = 0.01
iterations = 20
max_loss = 10.

img = gradient_ascent(img,
        iterations=iterations,
        step=step,
        max_loss=max_loss)

plt.rcParams["figure.figsize"] = (10,10)
plt.imshow(deprocess_image(img))
```

In [ ]:
```
import numpy as np
step = 0.01
num_octave = 3
octave_scale = 1.4
iterations = 20
max_loss = 10.
base_image_path = 'drive/My Drive/Colab Notebooks/hill_wiki.jpg'
img = preprocess_image(base_image_path)
```

In [ ]:
```
original_shape = img.shape[1:3]
successive_shapes = [original_shape]
for i in range(1, num_octave):
    shape = tuple([int(dim / (octave_scale ** i))
        for dim in original_shape])
    successive_shapes.append(shape)

successive_shapes = successive_shapes[::-1]
original_img = np.copy(img)
shrunk_original_img = resize_img(img, successive_shapes[0])
```

In [ ]:
```
successive_shapes
```

In [ ]:
```
#to make the effect look cooler
for shape in successive_shapes:
    print('Processing image shape', shape)
    #upscale image and apply gradient descent
    img = resize_img(img, shape)
    img = gradient_ascent(img,
        iterations=iterations,
        step=step,
        max_loss=max_loss)

    #insert details lost by upscaling
    upscaled_shrunk_original_img = resize_img(shrunk_original_img, shape) #will be pixelated
    same_size_original = resize_img(original_img, shape)                  #high-quality image by downscaling or
    lost_detail = same_size_original - upscaled_shrunk_original_img       #detail that was lost when scaling up

    img = img + lost_detail #reinsert the detail
    shrunk_original_img = resize_img(original_img, shape)
```

In [ ]:
```
plt.imshow(deprocess_image(img))
```

In [ ]: