TACSAY_HOA5_CPE313

April 18, 2024

1 Hands-on Activity 5.1 Generating Images with GANs and VAEs

Name: Tacsay, Marie Emmanuelle **Section:** CPE32S8 **Instructor:** Engr. Roman Richard

1.0.1 Pre-Lab Questions:

- **What are autoencoders? What is the vanilla/standard/regular architecture of an autoencoder? Answer:** Neural network designs called autoencoders are utilized for unsupervised learning tasks such as dimensionality reduction, denoising, and data compression. The encoder compresses input data into a lower-dimensional representation known as the latent space, and the decoder reconstructs it from this compressed form. They are composed of an encoder, decoder, and a bottleneck layer. The input data's dimensionality is progressively decreased by the encoder and then increased to its initial level by the decoder. The autoencoder gains the ability to effectively encode and decode input data through iterative training.
- **What are the limitations of autoencoders? Answer:** The limitations of autoencoders include overfitting, determinism, difficulties capturing intricate patterns, and the requirement for meticulous hyperparameter adjustment. Furthermore, they might not produce realistic samples as well as other models like GANs or VAEs as they are not naturally generative models. Additionally, they are prone to overfitting in small datasets and high-dimensional areas.
- **Why use variable autoencoders? Answer:** Variational Autoencoders (VAEs) are a versatile framework for learning latent representations and generating new data samples in an unsupervised manner. They generate diverse outputs by sampling from the learned latent space, providing a probabilistic framework for learning latent representations. VAEs also offer improved regularization by incorporating a prior distribution on the latent space, preventing overfitting and promoting structured and interpretable representations. This makes VAEs particularly useful for tasks requiring data generation or exploration.
- **What do we mean by generative adversarial networks? Answer:** In 2014, Ian Goodfellow created Generative Adversarial Networks (GANs), which are neural network topologies made up of a discriminator and a generator. The discriminator distinguishes between produced and true data, whereas the generator creates synthetic data samples. Through input from the discriminator, the generator improves its capacity to discern between authentic and fraudulent data by learning to generate realistic examples. GANs have demonstrated efficacy in producing realistic, high-quality data in a variety of fields.

#Procedures and Discussion: Variational Autoencoders and GANs

[ ]:
```
pip install imgaug
```

```
Collecting imgaug
  Downloading imgaug-0.4.0-py2.py3-none-any.whl.metadata (1.8 kB)
Requirement already satisfied: six in c:\users\emtac\anaconda3\lib\site-packages
(from imgaug) (1.16.0)
Requirement already satisfied: numpy>=1.15 in c:\users\emtac\anaconda3\lib\site-
packages (from imgaug) (1.24.3)
Requirement already satisfied: scipy in c:\users\emtac\anaconda3\lib\site-
packages (from imgaug) (1.11.4)
Requirement already satisfied: Pillow in c:\users\emtac\anaconda3\lib\site-
packages (from imgaug) (10.2.0)
Requirement already satisfied: matplotlib in c:\users\emtac\anaconda3\lib\site-
packages (from imgaug) (3.8.0)
Requirement already satisfied: scikit-image>=0.14.2 in
c:\users\emtac\anaconda3\lib\site-packages (from imgaug) (0.22.0)
Requirement already satisfied: opencv-python in
c:\users\emtac\anaconda3\lib\site-packages (from imgaug) (4.9.0.80)
Requirement already satisfied: imageio in c:\users\emtac\anaconda3\lib\site-
packages (from imgaug) (2.33.1)
Collecting Shapely (from imgaug)
  Downloading shapely-2.0.4-cp311-cp311-win_amd64.whl.metadata (7.2 kB)
Requirement already satisfied: networkx>=2.8 in
c:\users\emtac\anaconda3\lib\site-packages (from scikit-image>=0.14.2->imgaug)
(3.1)
Requirement already satisfied: tifffile>=2022.8.12 in
c:\users\emtac\anaconda3\lib\site-packages (from scikit-image>=0.14.2->imgaug)
(2023.4.12)
Requirement already satisfied: packaging>=21 in
c:\users\emtac\anaconda3\lib\site-packages (from scikit-image>=0.14.2->imgaug)
(23.1)
Requirement already satisfied: lazy_loader>=0.3 in
c:\users\emtac\anaconda3\lib\site-packages (from scikit-image>=0.14.2->imgaug)
(0.3)
Requirement already satisfied: contourpy>=1.0.1 in
c:\users\emtac\anaconda3\lib\site-packages (from matplotlib->imgaug) (1.2.0)
Requirement already satisfied: cycler>=0.10 in
c:\users\emtac\anaconda3\lib\site-packages (from matplotlib->imgaug) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
c:\users\emtac\anaconda3\lib\site-packages (from matplotlib->imgaug) (4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
c:\users\emtac\anaconda3\lib\site-packages (from matplotlib->imgaug) (1.4.4)
Requirement already satisfied: pyparsing>=2.3.1 in
c:\users\emtac\anaconda3\lib\site-packages (from matplotlib->imgaug) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in
c:\users\emtac\anaconda3\lib\site-packages (from matplotlib->imgaug) (2.8.2)
Downloading imgaug-0.4.0-py2.py3-none-any.whl (948 kB)
   ---------------------------------------- 0.0/948.0 kB ? eta -:--:--
```

```
   - -------------------------------------- 30.7/948.0 kB 660.6 kB/s eta 0:00:02
   ---------------------- ----------------- 553.0/948.0 kB 7.0 MB/s eta 0:00:01
   ---------------------------------------- 948.0/948.0 kB 8.6 MB/s eta 0:00:00
Downloading shapely-2.0.4-cp311-cp311-win_amd64.whl (1.4 MB)
   ---------------------------------------- 0.0/1.4 MB ? eta -:--:--
   ---------------- ---------------------- 0.6/1.4 MB 18.9 MB/s eta 0:00:01
   ---------------------------------------- 1.4/1.4 MB 22.7 MB/s eta 0:00:00
Installing collected packages: Shapely, imgaug
Successfully installed Shapely-2.0.4 imgaug-0.4.0
Note: you may need to restart the kernel to use updated packages.
```

[ ]:
```
# system libraries
import sys
import warnings
import os
import glob
warnings.filterwarnings("ignore")

# image libraries
import cv2 # requires installing opencv (e.g., pip install opencv-python)
from imgaug import augmenters # requires installing imgaug (e.g., pip install␣
 ↪imgaug)

# math/numerical libraries
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy
from scipy.stats import norm
from sklearn.model_selection import train_test_split

import tensorflow as tf

# deep learning libraries
from keras.models import Model, Sequential
from keras.optimizers import Adam, RMSprop

from keras.layers import *
from keras import backend as K
# from keras.callbacks import EarlyStopping
# from keras.utils import to_categorical
# from keras.metrics import *
# from keras.preprocessing import image, sequence
#

print(tf.__version__)
```

```
2.13.1
```

[ ]:
```
from tensorflow.python.framework.ops import disable_eager_execution
disable_eager_execution()
```

1.1 Part 1: Autoencoders (AEs)

Answer: What is the typical architecture of a 'vanilla/standard/traditional' autoencoder. **Answer:** The typical architecture of an autoencoder inludes encoder, decoder and bottleneck layer.

1.1.1 Data: Acquisition and Pre-processing

We will be using Fashion-MNIST, which can be conveniently accessed with Keras.

[ ]:
```
# get the data from keras - how convenient!
fashion_mnist = tf.keras.datasets.fashion_mnist

# load the data and split it into training and testing sets
(X_train, y_train),(X_test, y_test) = fashion_mnist.load_data()

# normalize the data by dividing with pixel intensity
# (each pixel is 8 bits so its value ranges from 0 to 255)
X_train, X_test = X_train / 255.0, X_test / 255.0

print(f'X_train shape: {X_train.shape}, X_test shape: {X_test.shape}')
print(f'y_train shape: {y_train.shape}, and y_test shape: {y_test.shape}')

# classes are named 0-9 so define names for plotting clarity
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# display the first 25 garments from the training set
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(X_train[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[y_train[i]])
plt.show()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 2us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 1s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0s/step
```

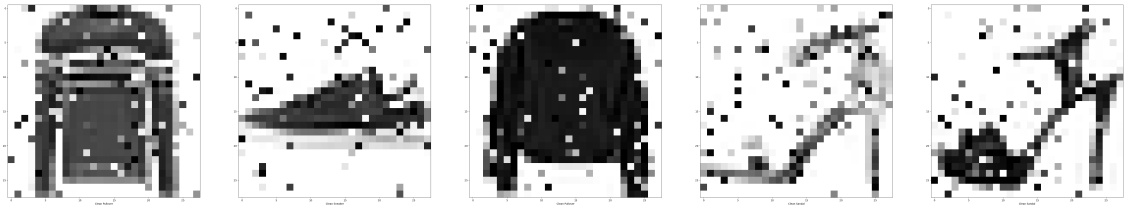
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
X_train shape: (60000, 28, 28), X_test shape: (10000, 28, 28)
y_train shape: (60000,), and y_test shape: (10000,)
```

Add Noise to the Images In attempt to make the autoencoder more robust and not just memorize the inputs, let's add noise to the inputs but calculate its loss based on how similar its outputs are to the original (non-denoised) images.

Check out imgaug docs for more info and other ways to add noise.

[ ]:
```
# Neural Networks want the inputs to be 3D
n_samples, h, w = X_train.shape

X_train = X_train.reshape(-1, h, w, 1)
X_test = X_test.reshape(-1, h, w, 1)
```

[ ]:
```
# Adding some 'salt and pepper' noises
noise = augmenters.SaltAndPepper(0.1)
seq_object = augmenters.Sequential([noise])

X_train_n = seq_object.augment_images(X_train * 255) / 255
X_test_n = seq_object.augment_images(X_test * 255) / 255

f, ax = plt.subplots(1,5)
f.set_size_inches(80, 40)
for i in range(5,10):
    ax[i-5].imshow(X_train_n[i, :, :, 0].reshape(28, 28), cmap=plt.cm.binary)
    ax[i-5].set_xlabel('Clean '+class_names[y_train[i]])
```



Create the Autoencoder

[ ]:
```
# input layer
input_layer = tf.keras.layers.Input(shape=(28, 28, 1))

# encoding architecture
encoded_layer1 = tf.keras.layers.Conv2D(64,(3, 3), activation='relu',␣
 ↪padding='same')(input_layer)
encoded_layer1 = tf.keras.layers.MaxPool2D((2, 2),␣
 ↪padding='same')(encoded_layer1)
encoded_layer2 = tf.keras.layers.Conv2D(32,(3, 3), activation='relu',␣
 ↪padding='same')(encoded_layer1)
encoded_layer2 = tf.keras.layers.MaxPool2D((2, 2),␣
 ↪padding='same')(encoded_layer2)
encoded_layer3 = tf.keras.layers.Conv2D(16,(3, 3), activation='relu',␣
 ↪padding='same')(encoded_layer2)
latent_view = tf.keras.layers.MaxPool2D((2, 2), padding='same')(encoded_layer3)

# decoding architecture
```

```
decoded_layer1 = tf.keras.layers.Conv2D(16, (3, 3), activation='relu',␣
 ↪padding='same')(latent_view)
decoded_layer1 = tf.keras.layers.UpSampling2D((2, 2))(decoded_layer1)
decoded_layer2 = tf.keras.layers.Conv2D(32, (3, 3), activation='relu',␣
 ↪padding='same')(decoded_layer1)
decoded_layer2 = tf.keras.layers.UpSampling2D((2, 2))(decoded_layer2)
decoded_layer3 = tf.keras.layers.Conv2D(64, (3, 3),␣
 ↪activation='relu')(decoded_layer2)
decoded_layer3 = tf.keras.layers.UpSampling2D((2, 2))(decoded_layer3)
output_layer = tf.keras.layers.Conv2D(1,(3, 3), padding='same')(decoded_layer3)

# compile the model
model = tf.keras.Model(input_layer, output_layer)
model.compile(optimizer='adam', loss='mse')
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d (Conv2D)             (None, 28, 28, 64)        640

 max_pooling2d (MaxPooling2  (None, 14, 14, 64)        0
 D)

 conv2d_1 (Conv2D)           (None, 14, 14, 32)        18464

 max_pooling2d_1 (MaxPoolin  (None, 7, 7, 32)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 7, 7, 16)          4624

 max_pooling2d_2 (MaxPoolin  (None, 4, 4, 16)          0
 g2D)

 conv2d_3 (Conv2D)           (None, 4, 4, 16)          2320

 up_sampling2d (UpSampling2  (None, 8, 8, 16)          0
 D)

 conv2d_4 (Conv2D)           (None, 8, 8, 32)          4640

 up_sampling2d_1 (UpSamplin  (None, 16, 16, 32)        0
 g2D)
```

```
 conv2d_5 (Conv2D)           (None, 14, 14, 64)        18496
=================================================================
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d (Conv2D)             (None, 28, 28, 64)        640

 max_pooling2d (MaxPooling2  (None, 14, 14, 64)        0
 D)

 conv2d_1 (Conv2D)           (None, 14, 14, 32)        18464

 max_pooling2d_1 (MaxPoolin  (None, 7, 7, 32)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 7, 7, 16)          4624

 max_pooling2d_2 (MaxPoolin  (None, 4, 4, 16)          0
 g2D)

 conv2d_3 (Conv2D)           (None, 4, 4, 16)          2320

 up_sampling2d (UpSampling2  (None, 8, 8, 16)          0
 D)

 conv2d_4 (Conv2D)           (None, 8, 8, 32)          4640

 up_sampling2d_1 (UpSamplin  (None, 16, 16, 32)        0
 g2D)

 conv2d_5 (Conv2D)           (None, 14, 14, 64)        18496

 up_sampling2d_2 (UpSamplin  (None, 28, 28, 64)        0
 g2D)

 conv2d_6 (Conv2D)           (None, 28, 28, 1)         577

=================================================================
Total params: 49761 (194.38 KB)
Trainable params: 49761 (194.38 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Train the Autoencoder

[ ]:
```
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',␣
↪min_delta=0, patience=10, verbose=5, mode='auto')
```

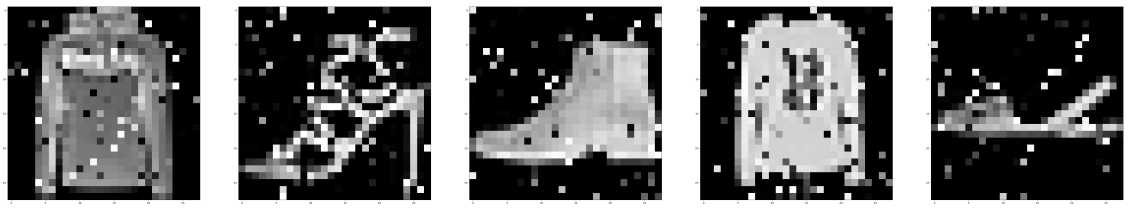
```
# epochs=20 for better results
history = model.fit(X_train_n, X_train, epochs=5, batch_size=2048,␣
 ↪validation_data=(X_test_n, X_test), callbacks=[early_stopping])
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
Epoch 1/5
60000/60000 [==============================] - 85s 1ms/sample - loss: 0.0797 -
val_loss: 0.0417
Epoch 2/5
60000/60000 [==============================] - 79s 1ms/sample - loss: 0.0373 -
val_loss: 0.0334
Epoch 3/5
60000/60000 [==============================] - 77s 1ms/sample - loss: 0.0306 -
val_loss: 0.0277
Epoch 4/5
60000/60000 [==============================] - 76s 1ms/sample - loss: 0.0260 -
val_loss: 0.0244
Epoch 5/5
60000/60000 [==============================] - 80s 1ms/sample - loss: 0.0235 -
val_loss: 0.0225
```

[ ]:
```
n = np.random.randint(0,len(X_test)-5) # pick a random starting index within␣
 ↪our test set
```

Visualize Samples reconstructed by AE Denoised Images

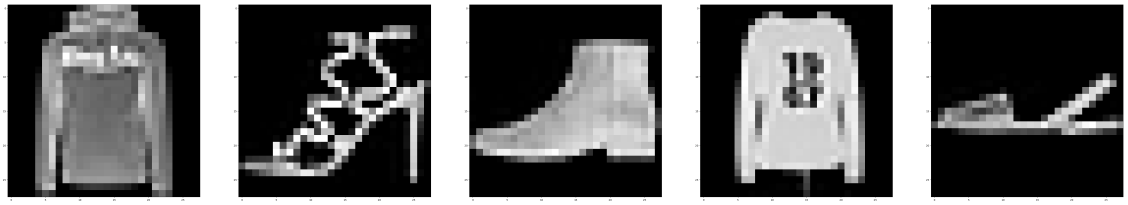
[ ]:
```
f, ax = plt.subplots(1,5)
f.set_size_inches(80, 40)
for i,a in enumerate(range(n,n+5)):
    ax[i].imshow(X_test_n[a, :, :, 0].reshape(28, 28), cmap='gray')
```



Actual Targets (i.e., Original inputs)

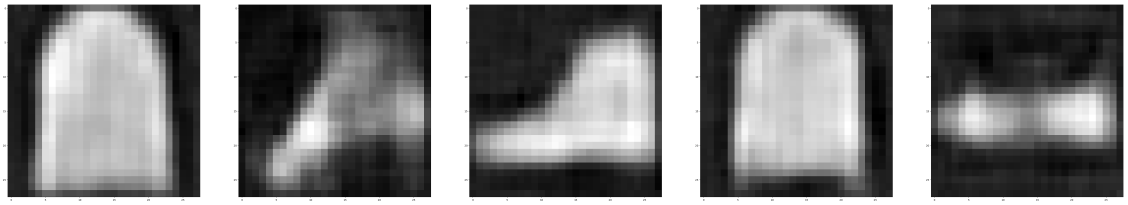
[ ]:
```
f, ax = plt.subplots(1,5)
f.set_size_inches(80, 40)
for i,a in enumerate(range(n,n+5)): # display the 5 images starting at our␣
 ↪random index
```

```
    ax[i].imshow(X_test[a, :, :, 0].reshape(28, 28), cmap='gray')
```



Predicted Images

[ ]:
```
preds = model.predict(X_test_n[n:n+5])
f, ax = plt.subplots(1,5)
f.set_size_inches(80, 40)
for i,a in enumerate(range(n,n+5)):
    ax[i].imshow(preds[i].reshape(28, 28), cmap='gray')
plt.show()
```



1.2 Part 2: Variational Autoencoders (VAEs)

Discussion: Why do we care to use VAEs? Identify the limitations/weaknesses of Autoencoders

Answer: VAE can be defined as being an autoencoder whose training is regularised to avoid overfitting and ensure that the latent space has good properties that enable generative processes. The limitations of autoencoders are it is computationally expensive especially when dealing large datasets and complex models. Also, autoencoders are prone to overfitting and may fail to capture complex data linkages when working with high-deminsional or structured data. Since autoencoders is usually used to work on models and images, training data in autoencoder can be computationally time-consuming especially for deep or intricate models.

1.2.1 Reset Data

[ ]:
```
# get the data from keras - how convenient!
fashion_mnist = tf.keras.datasets.fashion_mnist

# load the data and split it into training and testing sets
(X_train, y_train),(X_test, y_test) = fashion_mnist.load_data()
```

```
# normalize the data by dividing with pixel intensity
# (each pixel is 8 bits so its value ranges from 0 to 255)
X_train, X_test = X_train / 255.0, X_test / 255.0

print(f'X_train shape: {X_train.shape}, X_test shape: {X_test.shape}')
print(f'y_train shape: {y_train.shape}, and y_test shape: {y_test.shape}')
```

```
X_train shape: (60000, 28, 28), X_test shape: (10000, 28, 28)
y_train shape: (60000,), and y_test shape: (10000,)
```

1.2.2 Setup Encoder Neural Network

[ ]:
```
batch_size = 16
latent_dim = 2  # Number of latent dimension parameters

input_img = tf.keras.layers.Input(shape=(784,), name="input")
x = tf.keras.layers.Dense(512, activation='relu',␣
 ↪name="intermediate_encoder")(input_img)
x = tf.keras.layers.Dense(2, activation='relu', name="latent_encoder")(x)

z_mu = tf.keras.layers.Dense(latent_dim)(x)
z_log_sigma = tf.keras.layers.Dense(latent_dim)(x)
```

[ ]:
```
# sampling function
def sampling(args):
    z_mu, z_log_sigma = args
    epsilon = tf.keras.backend.random_normal(shape=(tf.keras.backend.
 ↪shape(z_mu)[0], latent_dim))
    z = z_mu + tf.keras.backend.exp(z_log_sigma) * epsilon
    return z

# sample vector from the latent distribution
z = tf.keras.layers.Lambda(sampling)([z_mu, z_log_sigma])
```

[ ]:
```
# decoder takes the latent distribution sample as input
decoder_input = tf.keras.layers.Input((2,), name="input_decoder")

x = tf.keras.layers.Dense(512, activation='relu', name="intermediate_decoder",␣
 ↪input_shape=(2,))(decoder_input)

# Expand to 784 total pixels
x = tf.keras.layers.Dense(784, activation='sigmoid', name="original_decoder")(x)

# decoder model statement
decoder = tf.keras.Model(decoder_input, x)
```

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

[ ]:
```
decoder.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_decoder (InputLayer)  [(None, 2)]               0

 intermediate_decoder (Dens  (None, 512)               1536
 e)

 original_decoder (Dense)    (None, 784)               402192

=================================================================
Total params: 403728 (1.54 MB)
Trainable params: 403728 (1.54 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_decoder (InputLayer)  [(None, 2)]               0

 intermediate_decoder (Dens  (None, 512)               1536
 e)

 original_decoder (Dense)    (None, 784)               402192

=================================================================
Total params: 403728 (1.54 MB)
Trainable params: 403728 (1.54 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

[ ]:
```
# construct a custom layer to calculate the loss
class CustomVariationalLayer(tf.keras.layers.Layer):

    def vae_loss(self, x, z_decoded):
        x = tf.keras.backend.flatten(x)
        z_decoded = tf.keras.backend.flatten(z_decoded)
        # Reconstruction loss
        xent_loss = tf.keras.losses.binary_crossentropy(x, z_decoded)
        return xent_loss

    # adds the 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

# apply the custom loss to the input images and the decoded latent distribution␣
 ↪sample
y = CustomVariationalLayer()([input_img, z_decoded])
```

[ ]:
```
z_decoded
```

[ ]:
```
<tf.Tensor 'model_1/original_decoder/Sigmoid:0' shape=(None, 784) dtype=float32>
```

[ ]:
```
# VAE model statement
vae = tf.keras.Model(input_img, y)
vae.compile(optimizer='rmsprop', loss=None)
```

```
WARNING:tensorflow:Output {0} 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_variational_layer.
```

[ ]:
```
vae.summary()
```

```
Model: "model_2"
__________________________________________________________________________________
__________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================
==================
 input (InputLayer)          [(None, 784)]             0         []

 intermediate_encoder (Dens  (None, 512)               401920
['input[0][0]']
 e)

 latent_encoder (Dense)      (None, 2)                 1026
['intermediate_encoder[0][0]']

 dense (Dense)               (None, 2)                 6
['latent_encoder[0][0]']

 dense_1 (Dense)             (None, 2)                 6
['latent_encoder[0][0]']

 lambda (Lambda)             (None, 2)                 0
['dense[0][0]',
```

```
'dense_1[0][0]']

 model_1 (Functional)        (None, 784)                  403728
['lambda[0][0]']

 custom_variational_layer (  (None, 784)                  0
['input[0][0]',
 CustomVariationalLayer)
'model_1[0][0]']

==================================================================================
==================
Total params: 806686 (3.08 MB)
Trainable params: 806686 (3.08 MB)
Non-trainable params: 0 (0.00 Byte)
----------------------------------------------------------------------------------
------------------
----------------------------------------------------------------------------------
------------------
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================
==================
 input (InputLayer)          [(None, 784)]                0         []

 intermediate_encoder (Dens  (None, 512)                  401920
['input[0][0]']
 e)

 latent_encoder (Dense)      (None, 2)                    1026
['intermediate_encoder[0][0]']

 dense (Dense)               (None, 2)                    6
['latent_encoder[0][0]']

 dense_1 (Dense)             (None, 2)                    6
['latent_encoder[0][0]']

 lambda (Lambda)             (None, 2)                    0
['dense[0][0]',
'dense_1[0][0]']

 model_1 (Functional)        (None, 784)                  403728
['lambda[0][0]']

 custom_variational_layer (  (None, 784)                  0
['input[0][0]',
 CustomVariationalLayer)
'model_1[0][0]']
```

```
=================================================================================
==================
Total params: 806686 (3.08 MB)
Trainable params: 806686 (3.08 MB)
Non-trainable params: 0 (0.00 Byte)
---------------------------------------------------------------------------------
------------------
```

[ ]:
```
X_train.shape
```

[ ]:
```
(60000, 28, 28)
```

[ ]:
```
train_x = X_train.reshape(-1,784) # train_x.reshape(-1, 784)
val_x = X_test.reshape(-1,784) #val_x.reshape(-1, 784)
```

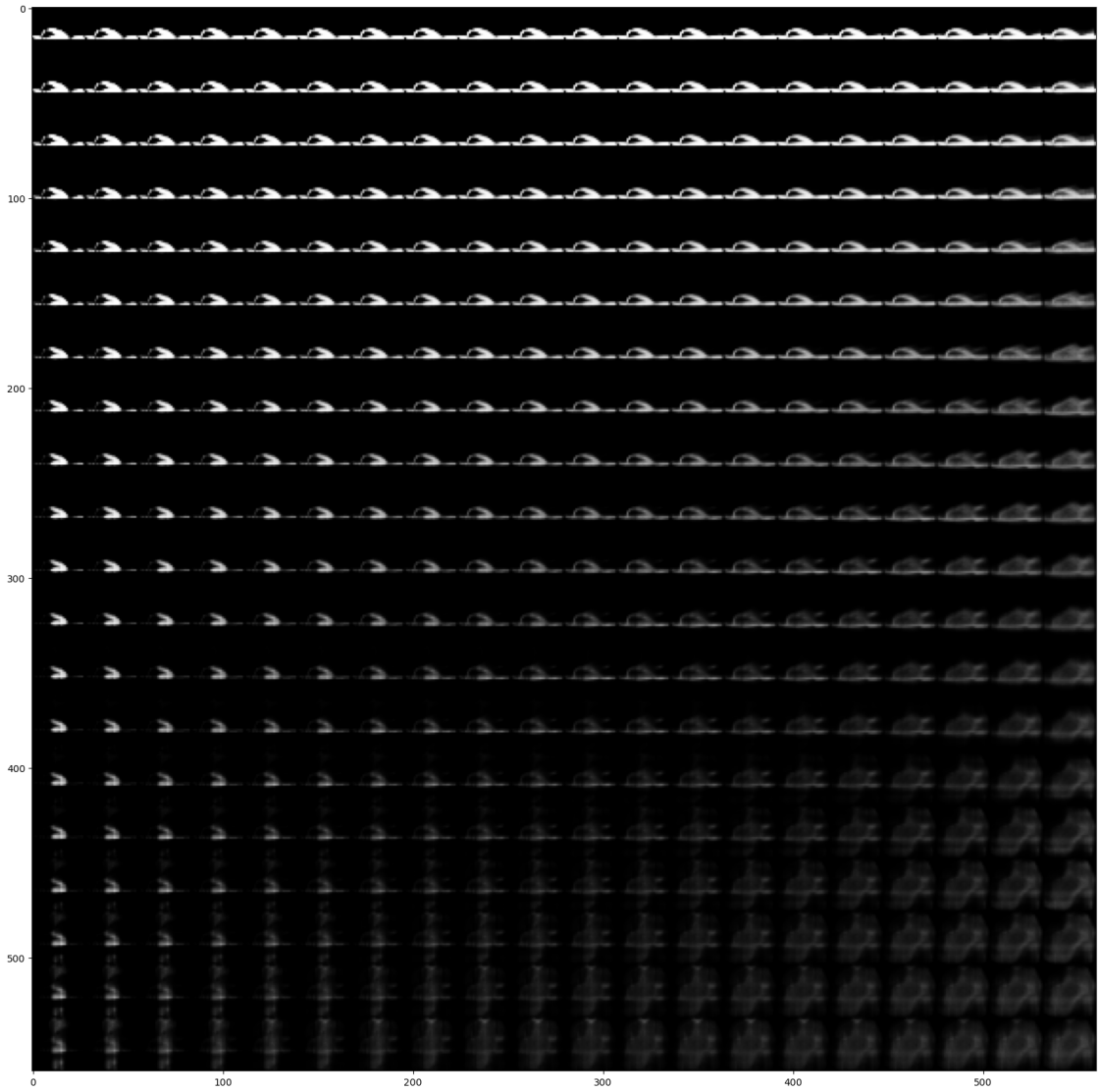
[ ]:
```
vae.fit(x=train_x, y=None,
        shuffle=True,
        epochs=4,
        batch_size=batch_size,
        validation_data=(val_x, None))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/4
60000/60000 [==============================] - 21s 346us/sample - loss: 0.3538 -
val_loss: 0.3393
Epoch 2/4
60000/60000 [==============================] - 21s 345us/sample - loss: 0.3363 -
val_loss: 0.3352
Epoch 3/4
60000/60000 [==============================] - 20s 327us/sample - loss: 0.3335 -
val_loss: 0.3329
Epoch 4/4
60000/60000 [==============================] - 20s 334us/sample - loss: 0.3319 -
val_loss: 0.3325
```

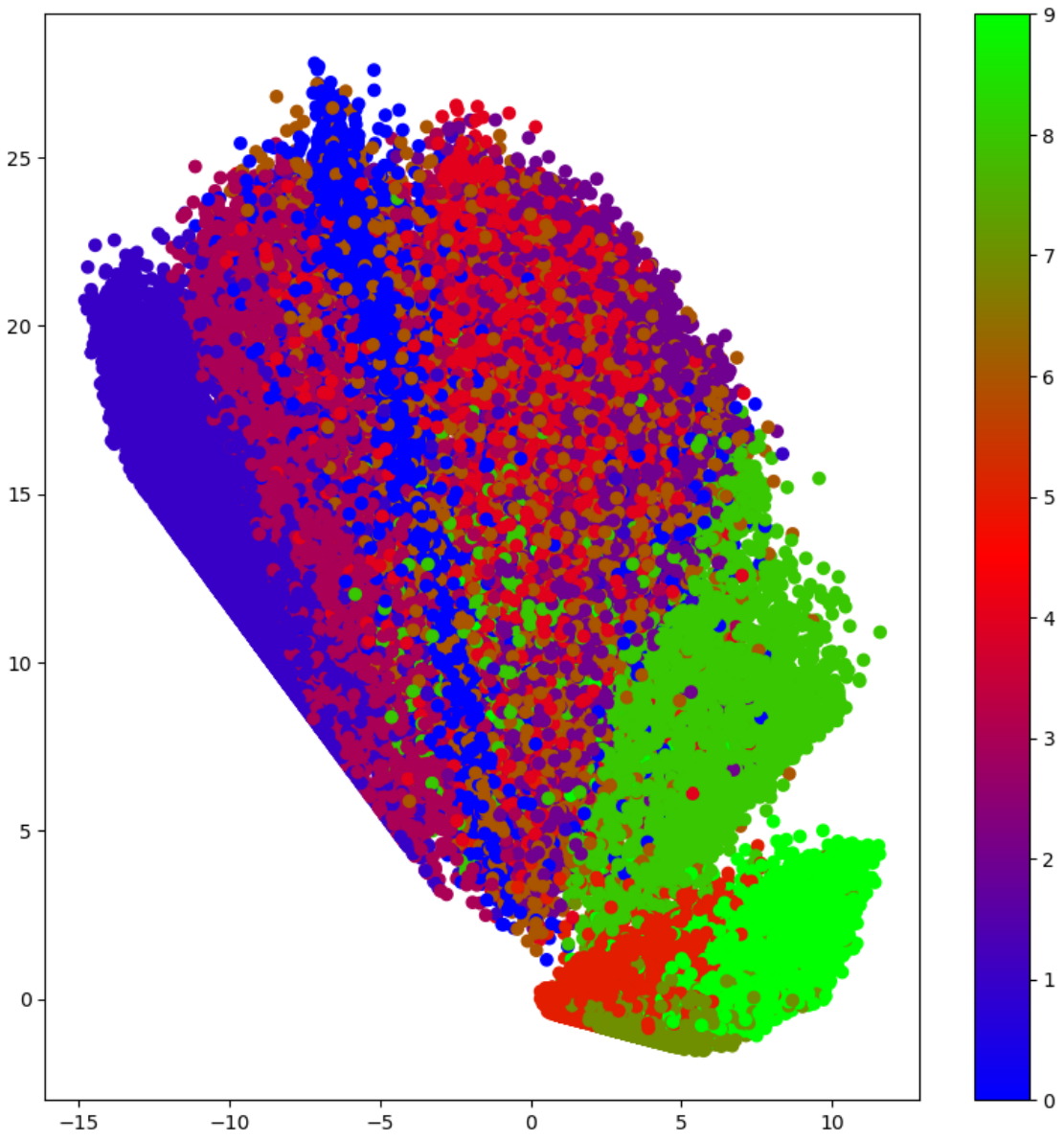
[ ]:
```
<keras.src.callbacks.History at 0x1a2cbb56850>
```

[ ]:
```
# Display a 2D manifold of the samples
n = 20  # figure with 20x20 samples
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))

# Construct grid of latent variable values - can change values here to generate␣
 ↪different things
grid_x = norm.ppf(np.linspace(0.05, 0.95, n))
grid_y = norm.ppf(np.linspace(0.05, 0.95, n))
```

```
# decode for each square in the grid
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=(20, 20))
plt.imshow(figure, cmap='gray')
plt.show()
```



[ ]:
```
# Translate into the latent space
encoder = tf.keras.Model(input_img, z_mu) # works on older version of TF and
 ↪Keras
x_valid_noTest_encoded = encoder.predict(train_x, batch_size=batch_size)
plt.figure(figsize=(10, 10))
plt.scatter(x_valid_noTest_encoded[:, 0], x_valid_noTest_encoded[:, 1],
 ↪c=y_train, cmap='brg')
plt.colorbar()
plt.show()
```



1.3 Part 2B: Adding CNNs and KL Divergence Losses

1.3.1 Generating new fashion

[ ]:
```
batch_size = 16
latent_dim = 2  # Number of latent dimension parameters

# Encoder architecture: Input -> Conv2D*4 -> Flatten -> Dense
input_img = Input(shape=(28, 28, 1))

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)

# need to know the shape of the network here for the decoder
shape_before_flattening = K.int_shape(x)

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

# Two outputs, latent mean and (log)variance
z_mu = Dense(latent_dim)(x)
z_log_sigma = Dense(latent_dim)(x)
```

1.3.2 Setup Sampling Function

[ ]:
```
# sampling function
def sampling(args):
    z_mu, z_log_sigma = args
    epsilon = K.random_normal(shape=(K.shape(z_mu)[0], latent_dim), mean=0.,␣
 ↪stddev=1.)
    return z_mu + K.exp(z_log_sigma) * epsilon

# sample vector from the latent distribution
z = Lambda(sampling)([z_mu, z_log_sigma])
```

1.3.3 Setup Decoder Neural Network

Task: Try different number of hidden layers and nodes?

[ ]:
```
# decoder takes the latent distribution sample as input
decoder_input = Input(K.int_shape(z)[1:])

# Expand to 784 total pixels
x = Dense(np.prod(shape_before_flattening[1:]),
                  activation='relu')(decoder_input)

# reshape
x = Reshape(shape_before_flattening[1:])(x)

# use Conv2DTranspose to reverse the conv layers from the encoder
x = Conv2DTranspose(32, 3,
                            padding='same',
                            activation='relu',
                            strides=(2, 2))(x)
x = Conv2D(1, 3,
                   padding='same',
```

```
                  activation='sigmoid')(x)

# decoder model statement
decoder = Model(decoder_input, x)

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

1.3.4 Setup Loss Function

[ ]:
```
# construct a custom layer to calculate the loss
class CustomVariationalLayer(Layer):

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

        # Reconstruction loss
        xent_loss = tf.keras.losses.binary_crossentropy(x, z_decoded)

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

    # adds the 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

# apply the custom loss to the input images and the decoded latent distribution␣
 ↪sample
y = CustomVariationalLayer()([input_img, z_decoded])
```

1.3.5 Train VAE

[ ]:
```
# VAE model statement
vae = Model(input_img, y)
vae.compile(optimizer='rmsprop', loss=None)
```

```
WARNING:tensorflow:Output {0} 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_variational_layer_1.
```

[ ]:
```
vae.summary()
```

```
Model: "model_5"
__________________________________________________________________________________
__________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================
==================
 input_2 (InputLayer)        [(None, 28, 28, 1)]       0         []

 conv2d_7 (Conv2D)           (None, 28, 28, 32)        320
['input_2[0][0]']

 conv2d_8 (Conv2D)           (None, 14, 14, 64)        18496
['conv2d_7[0][0]']

 conv2d_9 (Conv2D)           (None, 14, 14, 64)        36928
['conv2d_8[0][0]']

 conv2d_10 (Conv2D)          (None, 14, 14, 64)        36928
['conv2d_9[0][0]']

 flatten (Flatten)           (None, 12544)             0
['conv2d_10[0][0]']

 dense_2 (Dense)             (None, 32)                401440
['flatten[0][0]']

 dense_3 (Dense)             (None, 2)                 66
['dense_2[0][0]']

 dense_4 (Dense)             (None, 2)                 66
['dense_2[0][0]']

 lambda_1 (Lambda)           (None, 2)                 0
['dense_3[0][0]',
'dense_4[0][0]']

 model_4 (Functional)        (None, 28, 28, 1)         56385
['lambda_1[0][0]']

 custom_variational_layer_1  (None, 28, 28, 1)         0
['input_2[0][0]',
__________________________________________________________________________________
__________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================
==================
```

```
 input_2 (InputLayer)        [(None, 28, 28, 1)]          0         []

 conv2d_7 (Conv2D)           (None, 28, 28, 32)           320
['input_2[0][0]']

 conv2d_8 (Conv2D)           (None, 14, 14, 64)           18496
['conv2d_7[0][0]']

 conv2d_9 (Conv2D)           (None, 14, 14, 64)           36928
['conv2d_8[0][0]']

 conv2d_10 (Conv2D)          (None, 14, 14, 64)           36928
['conv2d_9[0][0]']

 flatten (Flatten)           (None, 12544)                0
['conv2d_10[0][0]']

 dense_2 (Dense)             (None, 32)                   401440
['flatten[0][0]']

 dense_3 (Dense)             (None, 2)                    66
['dense_2[0][0]']

 dense_4 (Dense)             (None, 2)                    66
['dense_2[0][0]']

 lambda_1 (Lambda)           (None, 2)                    0
['dense_3[0][0]',
'dense_4[0][0]']

 model_4 (Functional)        (None, 28, 28, 1)            56385
['lambda_1[0][0]']

 custom_variational_layer_1  (None, 28, 28, 1)            0
['input_2[0][0]',
  (CustomVariationalLayer)
'model_4[0][0]']

==================================================================================
==================
Total params: 550629 (2.10 MB)
Trainable params: 550629 (2.10 MB)
Non-trainable params: 0 (0.00 Byte)
----------------------------------------------------------------------------------
------------------
```

[ ]:
```
train_x = train_x.reshape(-1, 28, 28, 1)
val_x = val_x.reshape(-1, 28, 28, 1)
```

[ ]:
```
vae.fit(x=train_x, y=None,
        shuffle=True,
        epochs=20,
        batch_size=batch_size,
        validation_data=(val_x, None))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 126s 2ms/sample - loss: 0.3692 -
val_loss: 0.3392
Epoch 2/20
60000/60000 [==============================] - 120s 2ms/sample - loss: 0.3354 -
val_loss: 0.3352
Epoch 3/20
60000/60000 [==============================] - 122s 2ms/sample - loss: 0.3316 -
val_loss: 0.3323
Epoch 4/20
60000/60000 [==============================] - 124s 2ms/sample - loss: 0.3296 -
val_loss: 0.3298
Epoch 5/20
60000/60000 [==============================] - 128s 2ms/sample - loss: 0.3283 -
val_loss: 0.3302
Epoch 6/20
60000/60000 [==============================] - 133s 2ms/sample - loss: 0.3274 -
val_loss: 0.3285
Epoch 7/20
60000/60000 [==============================] - 134s 2ms/sample - loss: 0.3267 -
val_loss: 0.3283
Epoch 8/20
60000/60000 [==============================] - 133s 2ms/sample - loss: 0.3262 -
val_loss: 0.3314
Epoch 9/20
60000/60000 [==============================] - 137s 2ms/sample - loss: 0.3257 -
val_loss: 0.3285
Epoch 10/20
60000/60000 [==============================] - 132s 2ms/sample - loss: 0.3252 -
val_loss: 0.3286
Epoch 11/20
60000/60000 [==============================] - 148s 2ms/sample - loss: 0.3248 -
val_loss: 0.3265
Epoch 12/20
60000/60000 [==============================] - 143s 2ms/sample - loss: 0.3245 -
val_loss: 0.3275
Epoch 13/20
60000/60000 [==============================] - 138s 2ms/sample - loss: 0.3242 -
```

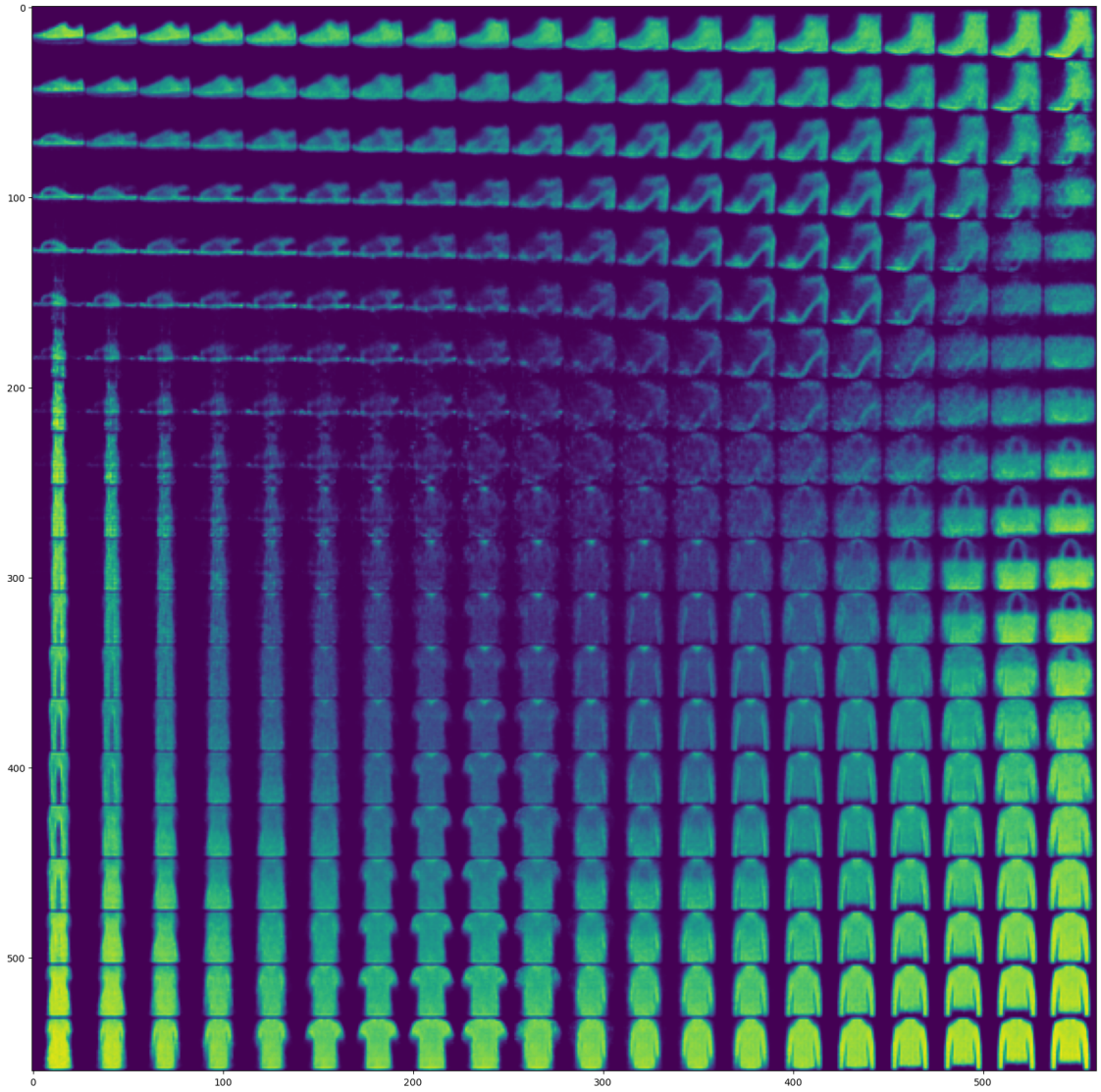
```
val_loss: 0.3271
Epoch 14/20
60000/60000 [==============================] - 135s 2ms/sample - loss: 0.3240 -
val_loss: 0.3256
Epoch 15/20
60000/60000 [==============================] - 136s 2ms/sample - loss: 0.3237 -
val_loss: 0.3262
Epoch 16/20
60000/60000 [==============================] - 132s 2ms/sample - loss: 0.3235 -
val_loss: 0.3262
Epoch 17/20
60000/60000 [==============================] - 136s 2ms/sample - loss: 0.3233 -
val_loss: 0.3255
Epoch 18/20
60000/60000 [==============================] - 137s 2ms/sample - loss: 0.3231 -
val_loss: 0.3256
Epoch 19/20
60000/60000 [==============================] - 137s 2ms/sample - loss: 0.3228 -
val_loss: 0.3261
Epoch 20/20
60000/60000 [==============================] - 139s 2ms/sample - loss: 0.3227 -
val_loss: 0.3250
```

[ ]:
```
<keras.src.callbacks.History at 0x1a2cbc5d510>
```

1.3.6 Visualize Samples Reconstructed by VAE

[ ]:
```
# Display a 2D manifold of the samples
n = 20  # figure with 20x20 samples
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))

# Construct grid of latent variable values - can change values here to generate␣
 ↪different things
grid_x = norm.ppf(np.linspace(0.05, 0.95, n))
grid_y = norm.ppf(np.linspace(0.05, 0.95, n))

# decode for each square in the grid
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
```

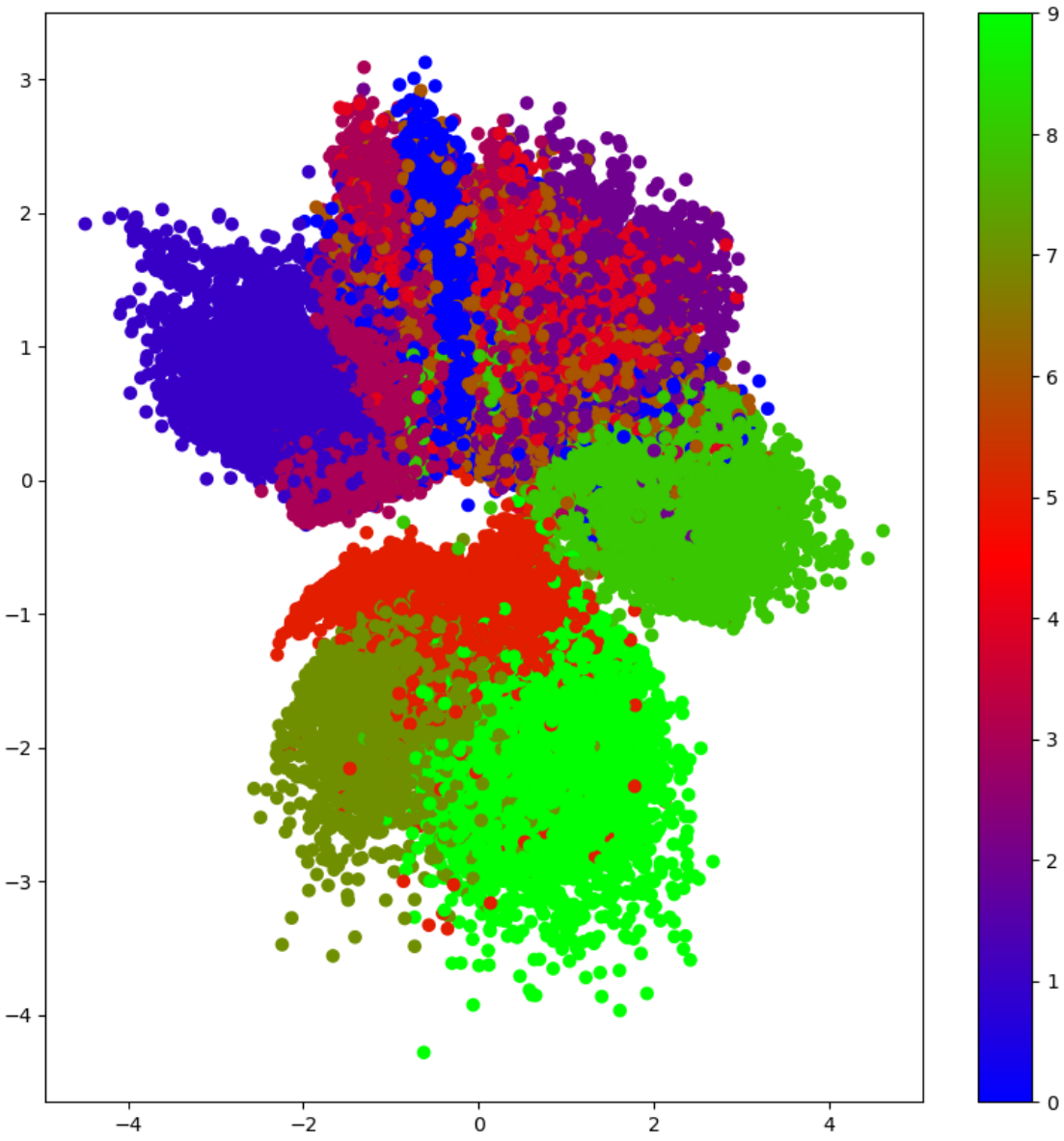
```
plt.figure(figsize=(20, 20))
plt.imshow(figure)
plt.show()
```



1.3.7 VAE: Visualize Latent Space

[ ]:
```
# Translate into the latent space
encoder = Model(input_img, z_mu)
x_valid_noTest_encoded = encoder.predict(train_x, batch_size=batch_size)
plt.figure(figsize=(10, 10))
plt.scatter(x_valid_noTest_encoded[:, 0], x_valid_noTest_encoded[:, 1],␣
 ↪c=y_train, cmap='brg')
plt.colorbar()
```

```
plt.show()
```



1.4 Exercise: Generating a 1D Gaussian Distribution from Uniform Noise

In this exercise, we are going to generate 1-D Gaussian distribution from a n-D uniform distribution. This is a toy exercise in order to understand the ability of GANs (generators) to generate arbitrary distributions from random noise.

Generate training data - Gaussian Distribution

[ ]:
```
# system libraries
import sys
import warnings
import os
import glob
warnings.filterwarnings("ignore")

# image libraries
import cv2 # requires installing opencv (e.g., pip install opencv-python)
from imgaug import augmenters # requires installing imgaug (e.g., pip install␣
 ↪imgaug)

# math/numerical libraries
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy
from scipy.stats import norm
from sklearn.model_selection import train_test_split

import tensorflow as tf
```

[ ]:
```
def generate_data(n_samples = 10000,n_dim=1):
  return np.random.randn(n_samples, n_dim)
```

A general function to define feedforward architectures

[ ]:
```
def set_model(input_dim, output_dim, hidden_dim=64,n_layers =␣
 ↪1,activation='tanh',optimizer='adam', loss = 'binary_crossentropy'):
  model = Sequential()
  model.add(Dense(hidden_dim,input_dim=input_dim,activation=activation))

  for _ in range(n_layers-1):
    model.add(Dense(hidden_dim),activation=activation)
  model.add(Dense(output_dim))

  model.compile(loss=loss, optimizer=optimizer)
  print(model.summary())
  return model
```

Setting GAN training and losses here

[ ]:
```
def get_gan_network(discriminator, random_dim, generator, optimizer = 'adam'):
  discriminator.trainable = False
  gan_input = Input(shape=(random_dim,))
  x = generator(gan_input)
  gan_output = discriminator(x)
  gan = Model(inputs = gan_input,outputs=gan_output)
```

```
  gan.compile( loss='binary_crossentropy', optimizer=optimizer)
  return gan
```

[ ]:
```
# hyper-parameters
NOISE_DIM = 10
DATA_DIM = 1
G_LAYERS = 1
D_LAYERS = 1
```

[ ]:
```
def train_gan(epochs=1,batch_size=128):
    x_train = generate_data(n_samples=12800,n_dim=DATA_DIM)
    batch_count = x_train.shape[0]/batch_size

    generator = set_model(NOISE_DIM, DATA_DIM, n_layers=G_LAYERS,␣
 ↪activation='tanh',loss = 'mean_squared_error')
    discriminator = set_model(DATA_DIM, 1, n_layers= D_LAYERS,␣
 ↪activation='sigmoid')
    gan = get_gan_network(discriminator, NOISE_DIM, generator, 'adam')

    for e in range(1,epochs+1):

        # generate noise from a uniform distribution
        noise = np.random.rand(batch_size,NOISE_DIM)
        true_batch = x_train[np.random.choice(x_train.shape[0], batch_size,␣
 ↪replace=False), :]

        generated_values = generator.predict(noise)
        X = np.concatenate([generated_values,true_batch])

        y_dis = np.zeros(2*batch_size)

        #One-sided label smoothing to avoid overconfidence. In GAN, if the␣
 ↪discriminator depends on a small set of features to detect real images,
        #the generator may just produce these features only to exploit the␣
 ↪discriminator.
        #The optimization may turn too greedy and produces no long term benefit.
        #To avoid the problem, we penalize the discriminator when the␣
 ↪prediction for any real images go beyond 0.9 (D(real image)>0.9).
        y_dis[:batch_size] = 0.9

        discriminator.trainable = True
        disc_history = discriminator.train_on_batch(X, y_dis)
        discriminator.trainable = False

        # Train generator
        noise = np.random.rand(batch_size,NOISE_DIM)
        y_gen = np.zeros(batch_size)
```

```
        gan.train_on_batch(noise, y_gen)

    return generator, discriminator
```

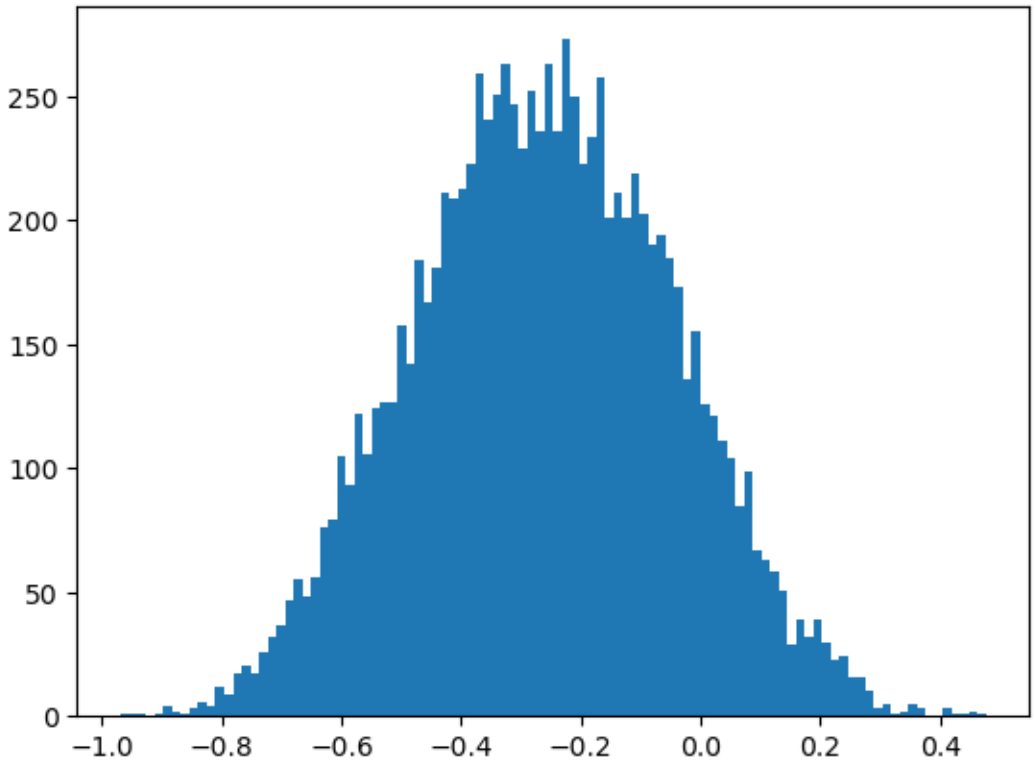
[ ]:
```
generator, discriminator = train_gan()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_6 (Dense)             (None, 64)                704

 dense_7 (Dense)             (None, 1)                 65

=================================================================
Total params: 769 (3.00 KB)
Trainable params: 769 (3.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_8 (Dense)             (None, 64)                128

 dense_9 (Dense)             (None, 1)                 65

=================================================================
Total params: 193 (772.00 Byte)
Trainable params: 193 (772.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

[ ]:
```
noise = np.random.rand(10000,NOISE_DIM)
generated_values = generator.predict(noise)
plt.hist(generated_values,bins=100)

true_gaussian = [np.random.randn() for x in range(10000)]

print('1st order moment - ', 'True : ', scipy.stats.moment(true_gaussian, 1) ,␣
 ↪', GAN :', scipy.stats.moment(generated_values,1))
print('2nd order moment - ', 'True : ', scipy.stats.moment(true_gaussian, 2) ,␣
 ↪', GAN :', scipy.stats.moment(generated_values,2))
print('3rd order moment - ', 'True : ', scipy.stats.moment(true_gaussian, 3) ,␣
 ↪', GAN :', scipy.stats.moment(generated_values,3))
```

```
print('4th order moment - ', 'True : ', scipy.stats.moment(true_gaussian, 4) ,␣
  ↪', GAN :', scipy.stats.moment(generated_values,4))
plt.show()
```

```
1st order moment -  True :  0.0 , GAN : [0.]
2nd order moment -  True :  0.9957432579222898 , GAN : [0.04679333]
3rd order moment -  True :  0.023517941979188665 , GAN : [0.00018427]
4th order moment -  True :  3.0957957196077803 , GAN : [0.00587803]
```



CONCLUSION

Some conclusions we can make from this exercise: 1. GANs are able to learn a generative model from general noise distributions. 2. Traditional GANs do not learn the higher-order moments well. Possible issues : Number of samples, approximating higher moments is hard. Usually known to under-predict higher order variances. For people interested in learning why, read more about divergence measures between distributions (particularly about Wasserstein etc.)

2 Supplementary Activities

2.1 Part 1: MNIST GAN - Learn to Generate MNIST Digits

[ ]:
```
pip install np_utils
```

```
Collecting np_utils
  Downloading np_utils-0.6.0.tar.gz (61 kB)
     ---------------------------------------- 0.0/62.0 kB ? eta -:--:--
     ---------- ----------------------------- 20.5/62.0 kB 640.0 kB/s eta 0:00:01
     ------------------------------------  61.4/62.0 kB 656.4 kB/s eta 0:00:01
     ---------------------------------------- 62.0/62.0 kB 663.4 kB/s eta 0:00:00
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Requirement already satisfied: numpy>=1.0 in c:\users\emtac\anaconda3\lib\site-
packages (from np_utils) (1.24.3)
Building wheels for collected packages: np_utils
  Building wheel for np_utils (setup.py): started
  Building wheel for np_utils (setup.py): finished with status 'done'
  Created wheel for np_utils: filename=np_utils-0.6.0-py3-none-any.whl
size=56454
sha256=12a2599f442743ba0014a115ce3b4c25596cb6db93b17bcbea437e38f3034fc4
  Stored in directory: c:\users\emtac\appdata\local\pip\cache\wheels\19\0d\33\ea
a4dcda5799bcbb51733c0744970d10edb4b9add4f41beb43
Successfully built np_utils
Installing collected packages: np_utils
Successfully installed np_utils-0.6.0
Note: you may need to restart the kernel to use updated packages.
```

[ ]:
```
from keras.datasets import mnist
import np_utils
from keras.models import Sequential, Model
from keras.layers import Input, Dense, Dropout, Activation, Flatten
from keras.layers import LeakyReLU
from keras.optimizers.legacy import Adam, RMSprop
import numpy as np
import matplotlib.pyplot as plt
import random
from tqdm import tqdm_notebook

# Dataset of 60,000 28x28 grayscale images of the 10 digits, along with a test␣
  ↪set of 10,000 images.
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
```

Re-scale data since we are using ReLU activations. **WHY? Answer:** We use ReLU activation to achieve non-liner transformation of the data. It is computationally inexpensive because it involves simple thresholding at zero. It allows networks to scale to many layers without significant increase in computational burden compared to more complex functions like tanh or sigmoid. Also, applying non-transformation is usually used in the hope that the transformed data will be close to linear

(regression) or close to linearly separable for classification. In addition, ReLU activation function is simple, fast to compute and doesn't suffer from vanishing gradients like sigmoid function.

[ ]:
```
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype('float32')/255
X_test = X_test.astype('float32')/255
```

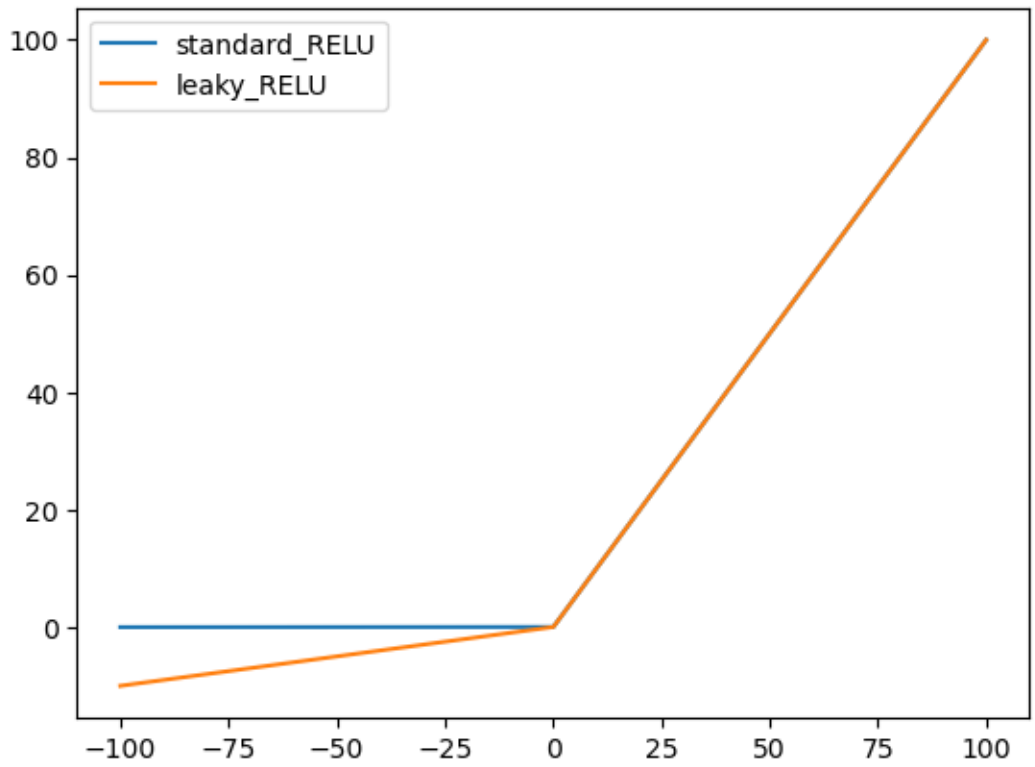
Set noise dimension

EXERCISE : Play around with different noise dimensions and plot the performance with respect to the size of the noise vector.

[ ]:
```
z_dim = 800
```

Tasks: 1. Build Model using LeakyReLU Activations: Build a generator, discriminator, and a GAN as feed-forward network with multiple layers, dropout, and leakyReLU as activation function. 2. Train Model 3. Provide an evaluation/conclusion.

[ ]:
```
def leakyReLU(x,neg_scale=0.01):
  if x > 0:
    return x
  else:
    return neg_scale*x

std_relu = []
leaky_relu = []

std_relu = [leakyReLU(x,neg_scale=0) for x in np.linspace(-100,100,10000)]
leaky_relu = [leakyReLU(x,neg_scale=0.1) for x in np.linspace(-100,100,10000)]

plt.plot(np.linspace(-100,100,10000),std_relu, label='standard_RELU')
plt.plot(np.linspace(-100,100,10000),leaky_relu, label='leaky_RELU')
plt.legend()
plt.show()
```



[ ]:
```
adam = Adam(learning_rate=0.0002, beta_1=0.5)


#GENERATOR
g = Sequential()
g.add(Dense(256, input_dim=z_dim, activation=LeakyReLU(alpha=0.2)))
g.add(Dense(512, activation=LeakyReLU(alpha=0.2)))
g.add(Dense(1024, activation=LeakyReLU(alpha=0.2)))
g.add(Dense(784, activation='sigmoid'))  # Values between 0 and 1
g.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])

#DISCRIMINATOR
d = Sequential()
d.add(Dense(1024, input_dim=784, activation=LeakyReLU(alpha=0.2)))
d.add(Dropout(0.3))
d.add(Dense(512, activation=LeakyReLU(alpha=0.2)))
d.add(Dropout(0.3))
d.add(Dense(256, activation=LeakyReLU(alpha=0.2)))
d.add(Dropout(0.3))
d.add(Dense(1, activation='sigmoid'))  # Values between 0 and 1
d.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])
```

```
#GAN
d.trainable = False
inputs = Input(shape=(z_dim, ))
hidden = g(inputs)
output = d(hidden)
gan = Model(inputs, output)
gan.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])
```

[ ]:
```
def plot_loss(losses):
    """
    @losses.keys():
        0: loss
        1: accuracy
    """
    d_loss = [v[0] for v in losses["D"]]
    g_loss = [v[0] for v in losses["G"]]

    plt.figure(figsize=(10,8))
    plt.plot(d_loss, label="Discriminator loss")
    plt.plot(g_loss, label="Generator loss")
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

def plot_generated(n_ex=10, dim=(1, 10), figsize=(12, 2)):
    noise = np.random.normal(0, 1, size=(n_ex, z_dim))
    generated_images = g.predict(noise)
    generated_images = generated_images.reshape(n_ex, 28, 28)

    plt.figure(figsize=figsize)
    for i in range(generated_images.shape[0]):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(generated_images[i], interpolation='nearest', cmap='gray_r')
        plt.axis('off')
    plt.tight_layout()
    plt.show()
```

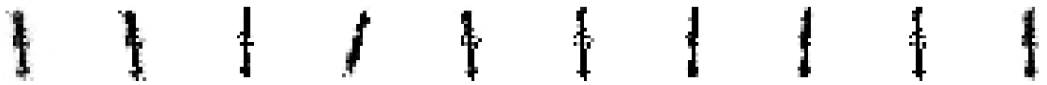
[ ]:
```
losses = {"D":[], "G":[]}

def train(epochs=1, plt_frq=1, BATCH_SIZE=128):
    batchCount = int(X_train.shape[0] / BATCH_SIZE)
    print('Epochs:', epochs)
    print('Batch size:', BATCH_SIZE)
    print('Batches per epoch:', batchCount)

    for e in tqdm_notebook(range(1, epochs+1)):
```

```
        if e == 1 or e%plt_frq == 0:
            print('-'*15, 'Epoch %d' % e, '-'*15)
        for _ in range(batchCount):  # tqdm_notebook(range(batchCount),␣
↪leave=False):
            # Create a batch by drawing random index numbers from the training␣
↪set
            image_batch = X_train[np.random.randint(0, X_train.shape[0],␣
↪size=BATCH_SIZE)]
            # Create noise vectors for the generator
            noise = np.random.normal(0, 1, size=(BATCH_SIZE, z_dim))

            # Generate the images from the noise
            generated_images = g.predict(noise)
            X = np.concatenate((image_batch, generated_images))
            # Create labels
            y = np.zeros(2*BATCH_SIZE)
            y[:BATCH_SIZE] = 0.9  # One-sided label smoothing

            # Train discriminator on generated images
            d.trainable = True
            d_loss = d.train_on_batch(X, y)

            # Train generator
            noise = np.random.normal(0, 1, size=(BATCH_SIZE, z_dim))
            y2 = np.ones(BATCH_SIZE)
            d.trainable = False
            g_loss = gan.train_on_batch(noise, y2)

        # Only store losses from final
        losses["D"].append(d_loss)
        losses["G"].append(g_loss)

        # Update the plots
        if e == 1 or e%plt_frq == 0:
            plot_generated()
    plot_loss(losses)
```

[ ]:
```
train(epochs=100, plt_frq=10, BATCH_SIZE=128)
```

```
Epochs: 100
Batch size: 128
Batches per epoch: 468
  0%|          | 0/100 [00:00<?, ?it/s]
--------------- Epoch 1 ---------------
```

---------- Epoch 10 ----------



---------- Epoch 20 ----------

---------- Epoch 30 ----------

---------- Epoch 40 ----------

---------- Epoch 50 ----------

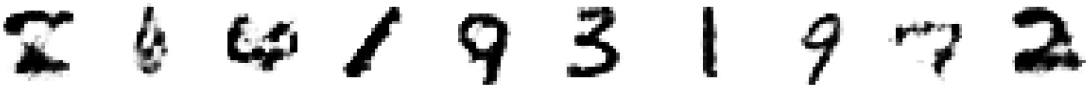
---------- Epoch 60 ----------

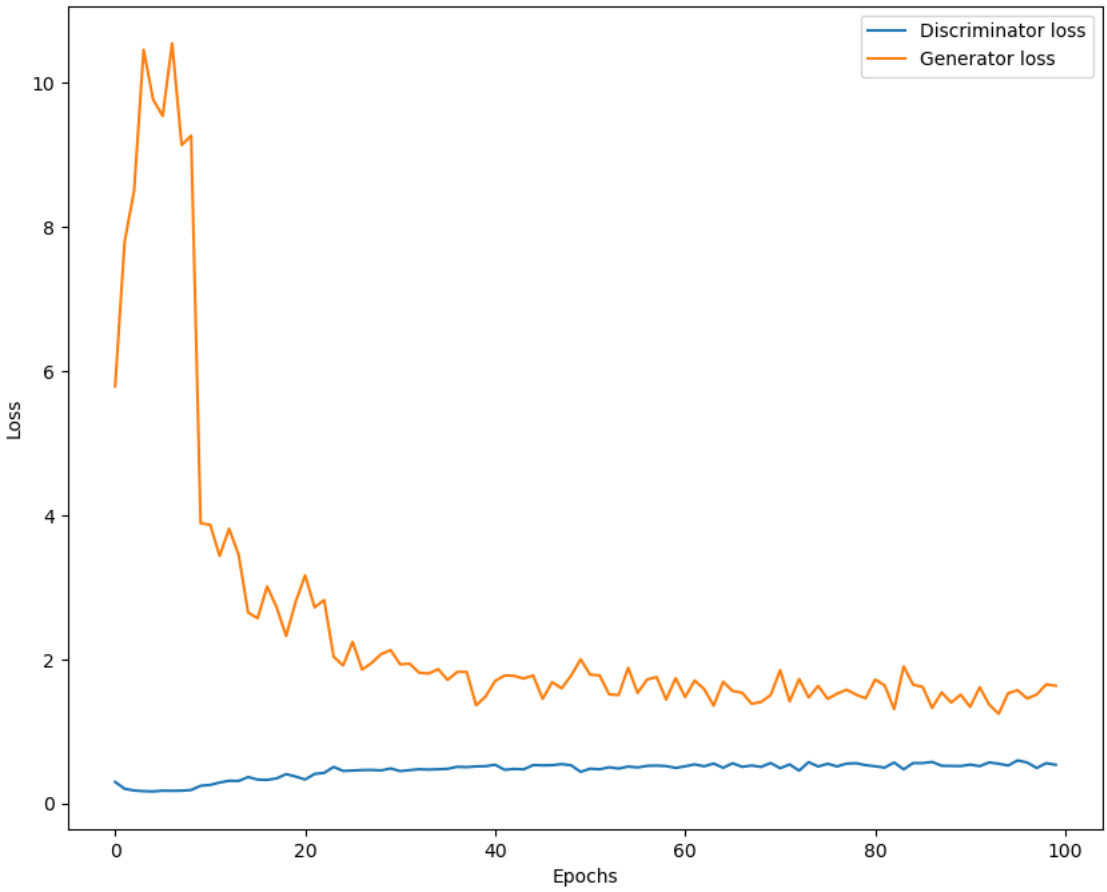
---------- Epoch 70 ----------

---------- Epoch 80 ----------

---------- Epoch 90 ----------

---------- Epoch 100 ----------





Aanalysis: In this part of the activity, we were tasked to create GAN model to generate digits from the MNIST dataset. In this case, the size of noise vector was changed into 800 as well as the epochs and plot frequency. The resulting output shows that as the epochs increases, the model will generate a much clearer output of images containing the digits.

2.2 Part 2: GANs and VAEs

Tasks: 1. Use your own dataset. 2. Generate new images using GAN and VAE 3. Compare the results of generating images using GAN and VAE.

[ ]:
```
import tensorflow as tf
from tensorflow import keras
K = keras.backend

import numpy as np
import matplotlib.pyplot as plt
import random
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
```

```
BATCH_SIZE = 32

IMG_WIDTH  = 48
IMG_HEIGHT = 48

KERNEL_SIZE = 4
ENCODING_SIZE = 32
```

[ ]:
```
# create a training data generator
training_generator = tf.keras.preprocessing.image_dataset_from_directory(
  r"C:\Users\emtac\Downloads\Butterfly\MONARCH",
  seed = 42,
  image_size = (IMG_HEIGHT, IMG_WIDTH),
  batch_size = BATCH_SIZE,
  labels = None,
  color_mode = 'rgb'
)

training_generator = training_generator.prefetch(1)
```

```
Found 128 files.
```

[ ]:
```
generator = keras.Sequential([
  keras.layers.Dense(256 * 6 * 6, activation = "selu", input_shape =␣
 ↪[ENCODING_SIZE]),
  keras.layers.Reshape([6, 6, 256]),
  keras.layers.BatchNormalization(),
  keras.layers.Conv2DTranspose(filters = 128, kernel_size = KERNEL_SIZE,␣
 ↪strides = 2,
                               padding = "same", activation = "selu",
                               kernel_initializer='lecun_normal'),
  keras.layers.BatchNormalization(),
  keras.layers.Conv2DTranspose(filters = 64, kernel_size = KERNEL_SIZE, strides␣
 ↪= 2,
                               padding = "same", activation = "selu",
                               kernel_initializer='lecun_normal'),
  keras.layers.BatchNormalization(),
  keras.layers.Conv2DTranspose(filters = 32, kernel_size = KERNEL_SIZE, strides␣
 ↪= 2,
                               padding = "same", activation = "selu",
                               kernel_initializer='lecun_normal'),
  keras.layers.Conv2DTranspose(filters = 3, kernel_size = KERNEL_SIZE, strides␣
 ↪= 1,
                               padding = 'same', activation = 'sigmoid')
])
generator.summary()
```

```
discriminator = keras.Sequential([
  keras.layers.Conv2D(32, input_shape = [IMG_WIDTH, IMG_HEIGHT, 3], kernel_size␣
 ↪= KERNEL_SIZE,
                      strides = 1, padding = 'same', activation = keras.layers.
 ↪LeakyReLU(0.2)),
  keras.layers.Dropout(0.25),
  keras.layers.Conv2D(64, kernel_size = KERNEL_SIZE, strides = 2,
                      padding = 'same', activation = keras.layers.LeakyReLU(0.
 ↪2)),
  keras.layers.Dropout(0.25),
  keras.layers.Conv2D(128, kernel_size = KERNEL_SIZE, strides = 2,
                      padding = 'same', activation = keras.layers.LeakyReLU(0.
 ↪2)),
  keras.layers.Dropout(0.25),
  keras.layers.Conv2D(256, kernel_size = KERNEL_SIZE, strides = 2,
                      padding = 'same', activation = keras.layers.LeakyReLU(0.
 ↪2)),
  keras.layers.Dropout(0.25),
  keras.layers.Flatten(),
  keras.layers.Dense(1, activation = 'sigmoid')
])
discriminator.summary()

gan = keras.models.Sequential([generator, discriminator])
gan.summary()
```

```
c:\Users\emtac\AppData\Local\Programs\Python\Python312\Lib\site-
packages\keras\src\layers\core\dense.py:85: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 9216)	304,128
reshape (Reshape)	(None, 6, 6, 256)	0
batch_normalization (BatchNormalization)	(None, 6, 6, 256)	1,024
conv2d_transpose (Conv2DTranspose)	(None, 12, 12, 128)	524,416

batch_normalization_1 (BatchNormalization)	(None, 12, 12, 128)	512
conv2d_transpose_1 (Conv2DTranspose)	(None, 24, 24, 64)	131,136
batch_normalization_2 (BatchNormalization)	(None, 24, 24, 64)	256
conv2d_transpose_2 (Conv2DTranspose)	(None, 48, 48, 32)	32,800
conv2d_transpose_3 (Conv2DTranspose)	(None, 48, 48, 3)	1,539

Total params: 995,811 (3.80 MB)

Trainable params: 994,915 (3.80 MB)

Non-trainable params: 896 (3.50 KB)

```
c:\Users\emtac\AppData\Local\Programs\Python\Python312\Lib\site-
packages\keras\src\layers\convolutional\base_conv.py:99: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 48, 48, 32)	1,568
dropout (Dropout)	(None, 48, 48, 32)	0
conv2d_1 (Conv2D)	(None, 24, 24, 64)	32,832
dropout_1 (Dropout)	(None, 24, 24, 64)	0
conv2d_2 (Conv2D)	(None, 12, 12, 128)	131,200
dropout_2 (Dropout)	(None, 12, 12, 128)	0

```
 conv2d_3 (Conv2D)                (None, 6, 6, 256)              524,544

 dropout_3 (Dropout)              (None, 6, 6, 256)                    0

 flatten (Flatten)                (None, 9216)                         0

 dense_1 (Dense)                  (None, 1)                        9,217


 Total params: 699,361 (2.67 MB)

 Trainable params: 699,361 (2.67 MB)

 Non-trainable params: 0 (0.00 B)
```

```
Model: "sequential_2"
```

```
 Layer (type)                     Output Shape                   Param #

 sequential (Sequential)          ?                              995,811

 sequential_1 (Sequential)        ?                              699,361


 Total params: 1,695,172 (6.47 MB)

 Trainable params: 1,694,276 (6.46 MB)

 Non-trainable params: 896 (3.50 KB)
```

[ ]:
```python
def plot_multiple_images(images, n_cols=None):
    n_cols = n_cols or len(images)
    n_rows = (len(images) - 1) // n_cols + 1

    if images.shape[-1] == 1:
        images = np.squeeze(images, axis=-1)
    plt.figure(figsize=(n_cols * 3, n_rows * 3))

    for index, image in enumerate(images):
        plt.subplot(n_rows, n_cols, index + 1)
```

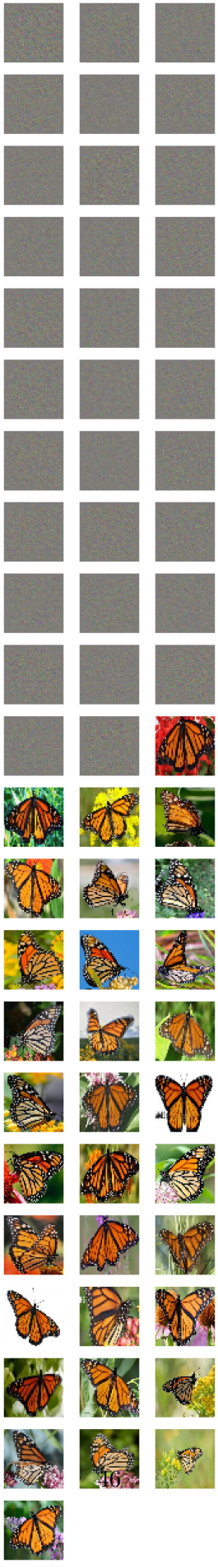
```
        plt.imshow(image, cmap="binary")
        plt.axis("off")

def train_gan(gan, dataset, BATCH_SIZE, ENCODING_SIZE, n_epochs = 50,␣
 ↪plot_frequency = 10, learning_rate = 0.01):
    generator, discriminator = gan.layers
    for epoch in range(n_epochs):
        print("Epoch {}/{}".format(epoch + 1, n_epochs))
        preds = []
        actuals = []
        for X_batch in dataset:
          print("=", end = '')
          X_batch /= 255

          # phase 1 - training the discriminator
          noise = tf.random.normal(shape=[len(X_batch), ENCODING_SIZE])
          generated_images = generator(noise)
          X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
          y1 = tf.constant([[0.]] * len(X_batch) + [[1.]] * len(X_batch))
          y1 += 0.05 * tf.random.uniform(tf.shape(y1)) # add random noise to␣
 ↪labels
          discriminator.trainable = True
          discriminator.train_on_batch(X_fake_and_real, y1)

          # capture the actual and predicted values for later
          actuals.extend([y.numpy()[0] for y in y1])
          preds.extend([y[0] for y in discriminator.predict(X_fake_and_real).
 ↪tolist()])

          # phase 2 - training the generator
          noise = tf.random.normal(shape=[len(X_batch), ENCODING_SIZE])
          y2 = tf.constant([[1.]] * len(X_batch))
          discriminator.trainable = False
          gan.train_on_batch(noise, y2)

        # print out a confusion matrix to see how the discriminator is doing
        print()
        print(confusion_matrix([1 if a > 0.5 else 0 for a in actuals],
                               [1 if p > 0.5 else 0 for p in preds]))
        print('accuracy: ', accuracy_score([1 if a > 0.5 else 0 for a in␣
 ↪actuals],
                                           [1 if p > 0.5 else 0 for p in␣
 ↪preds]))

        # only plot every 10 epochs
        if epoch % plot_frequency == 0:
          plot_multiple_images(X_fake_and_real, 3)
```

```
        plt.show()

    plot_multiple_images(X_fake_and_real, 3)
    plt.show()
```

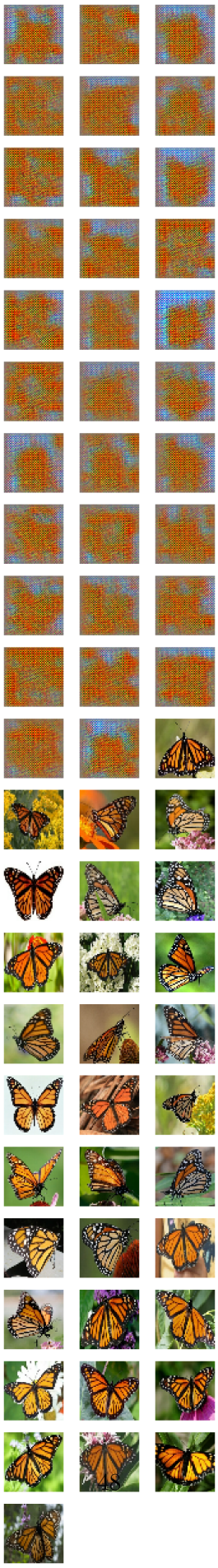
[ ]:
```
# clear the session for a clean run
keras.backend.clear_session()
tf.random.set_seed(42)

discriminator.compile(loss = 'binary_crossentropy', optimizer = keras.
 ↪optimizers.RMSprop())
discriminator.trainable = False
gan.compile(loss = 'binary_crossentropy', optimizer = keras.optimizers.
 ↪RMSprop())

train_gan(gan, training_generator, BATCH_SIZE, ENCODING_SIZE, n_epochs = 20,␣
 ↪plot_frequency = 5)
```

```
WARNING:tensorflow:From
c:\Users\emtac\AppData\Local\Programs\Python\Python312\Lib\site-
packages\keras\src\backend\common\global_state.py:73: The name
tf.reset_default_graph is deprecated. Please use
tf.compat.v1.reset_default_graph instead.

Epoch 1/20
2/2             0s 8ms/step
2/2             0s 8ms/step
=WARNING:tensorflow:5 out of the last 5 calls to <function
TensorFlowTrainer.make_train_function.<locals>.one_step_on_iterator at
0x000002A869651DA0> triggered tf.function retracing. Tracing is expensive and
the excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has reduce_retracing=True option that
can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
2/2             0s 8ms/step
WARNING:tensorflow:6 out of the last 6 calls to <function
TensorFlowTrainer.make_train_function.<locals>.one_step_on_iterator at
0x000002A86CABD8A0> triggered tf.function retracing. Tracing is expensive and
the excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has reduce_retracing=True option that
can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
```

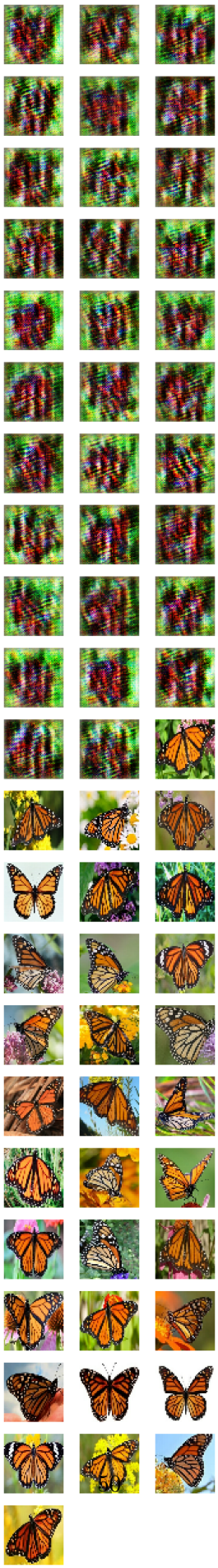
```
2/2             0s 9ms/step

[[96 32]
 [72 56]]
accuracy:  0.59375
```



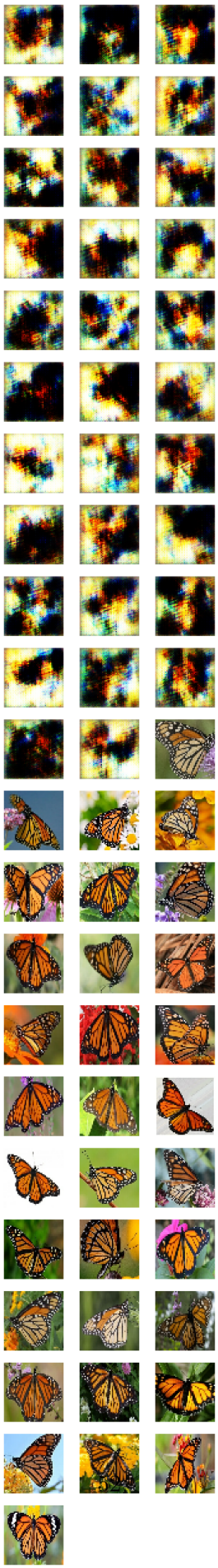
```
Epoch 2/20
2/2             0s 9ms/step
2/2             0s 8ms/step
2/2             0s 7ms/step
2/2             0s 10ms/step

[[ 47  81]
 [ 13 115]]
accuracy:  0.6328125
Epoch 3/20
2/2             0s 7ms/step
2/2             0s 8ms/step
2/2             0s 10ms/step
2/2             0s 9ms/step

[[ 50  78]
 [  7 121]]
accuracy:  0.66796875
Epoch 4/20
2/2             0s 7ms/step
2/2             0s 8ms/step
2/2             0s 8ms/step
2/2             0s 9ms/step

[[111  17]
 [ 55  73]]
accuracy:  0.71875
Epoch 5/20
2/2             0s 7ms/step
2/2             0s 7ms/step
2/2             0s 8ms/step
2/2             0s 7ms/step

[[96 32]
 [65 63]]
accuracy:  0.62109375
Epoch 6/20
2/2             0s 7ms/step
2/2             0s 7ms/step
2/2             0s 7ms/step
2/2             0s 8ms/step

[[128   0]
 [ 63  65]]
accuracy:  0.75390625
```



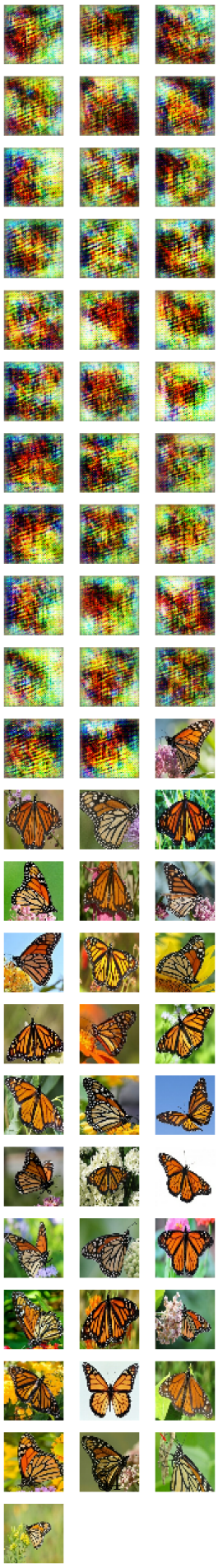
```
Epoch 7/20
2/2             0s 8ms/step
2/2             0s 9ms/step
2/2             0s 8ms/step
2/2             0s 7ms/step

[[64 64]
 [56 72]]
accuracy:  0.53125
Epoch 8/20
2/2             0s 8ms/step
2/2             0s 9ms/step
2/2             0s 8ms/step
2/2             0s 8ms/step

[[ 46  82]
 [  0 128]]
accuracy:  0.6796875
Epoch 9/20
2/2             0s 7ms/step
2/2             0s 7ms/step
2/2             0s 7ms/step
2/2             0s 9ms/step

[[112  16]
 [ 38  90]]
accuracy:  0.7890625
Epoch 10/20
2/2             0s 9ms/step
2/2             0s 8ms/step
2/2             0s 7ms/step
2/2             0s 12ms/step

[[92 36]
 [62 66]]
accuracy:  0.6171875
Epoch 11/20
2/2             0s 8ms/step
2/2             0s 9ms/step
2/2             0s 10ms/step
2/2             0s 9ms/step

[[ 74  54]
 [ 25 103]]
accuracy:  0.69140625
```



```
Epoch 12/20
2/2             0s 7ms/step
2/2             0s 8ms/step
2/2             0s 7ms/step
2/2             0s 7ms/step

[[61 67]
 [31 97]]
accuracy:  0.6171875
Epoch 13/20
2/2             0s 7ms/step
2/2             0s 9ms/step
2/2             0s 8ms/step
2/2             0s 7ms/step

[[112  16]
 [ 48  80]]
accuracy:  0.75
Epoch 14/20
2/2             0s 7ms/step
2/2             0s 7ms/step
2/2             0s 8ms/step
2/2             0s 8ms/step

[[ 86  42]
 [  9 119]]
accuracy:  0.80078125
Epoch 15/20
2/2             0s 8ms/step
2/2             0s 9ms/step
2/2             0s 9ms/step
2/2             0s 8ms/step

[[62 66]
 [33 95]]
accuracy:  0.61328125
Epoch 16/20
2/2             0s 7ms/step
2/2             0s 7ms/step
2/2             0s 7ms/step
2/2             0s 7ms/step

[[ 70  58]
 [  0 128]]
accuracy:  0.7734375
```



```
Epoch 17/20
2/2             0s 7ms/step
2/2             0s 7ms/step
2/2             0s 8ms/step
2/2             0s 8ms/step

[[ 67  61]
 [  2 126]]
accuracy:  0.75390625
Epoch 18/20
2/2             0s 8ms/step
2/2             0s 7ms/step
2/2             0s 7ms/step
2/2             0s 7ms/step

[[ 69  59]
 [ 25 103]]
accuracy:  0.671875
Epoch 19/20
2/2             0s 8ms/step
2/2             0s 7ms/step
2/2             0s 7ms/step
2/2             0s 7ms/step

[[70 58]
 [38 90]]
accuracy:  0.625
Epoch 20/20
2/2             0s 7ms/step
2/2             0s 7ms/step
2/2             0s 9ms/step
2/2             0s 7ms/step

[[81 47]
 [36 92]]
accuracy:  0.67578125
```



Analysis: In this part of the activity, it shows how GAN work on the dataset chosen which us the butterfly dataset. Creation of discriminator and generator were shown in order for the GAN to fully demonstrated. Also, the output shows how GAN generate the original data as the epochs increase.

2.2.1 VAE

[ ]:
```
import tensorflow as tf
from tensorflow import keras
K = keras.backend

import numpy as np
import matplotlib.pyplot as plt
import random

BATCH_SIZE = 32

IMG_WIDTH  = 96
IMG_HEIGHT = 96

ENCODING_SIZE = 8
```

[ ]:
```
# create a butterfly with labels generator
butterfly_generator = tf.keras.preprocessing.image_dataset_from_directory(
  r"C:\Users\emtac\Downloads\Butterfly",
  seed = 84,
  image_size = (IMG_HEIGHT, IMG_WIDTH),
  batch_size = BATCH_SIZE,
  labels = 'inferred'
)

# holds the butterfly names
butterfly_names = butterfly_generator.class_names

# create a training data generator
training_generator = tf.keras.preprocessing.image_dataset_from_directory(
  r"C:\Users\emtac\Downloads\Butterfly",
  seed = 42,
  image_size = (IMG_HEIGHT, IMG_WIDTH),
  batch_size = BATCH_SIZE,
  labels = None,
  validation_split = 0.2,
  subset = 'training'
)
```

```
# and a validation set generator
validation_generator = tf.keras.preprocessing.image_dataset_from_directory(
  r"C:\Users\emtac\Downloads\Butterfly",
  seed = 42,
  image_size = (IMG_HEIGHT, IMG_WIDTH),
  batch_size = BATCH_SIZE,
  labels = None,
  validation_split = 0.2,
  subset = 'validation'
)

# autoencoders require the input matrix as both the input
# and the output
# provide input -> input for the autoencoder
def replicate_inputs_to_outputs(images):
  return images/255, images/255

# create prefetch generators to speed things up a bit
training_generator = training_generator.prefetch(128)
validation_generator = validation_generator.prefetch(128)

# create the final generators to be used in training the autoencoders
X_train = training_generator.map(replicate_inputs_to_outputs)
X_valid = validation_generator.map(replicate_inputs_to_outputs)
```

```
Found 1179 files belonging to 10 classes.
```

```
Found 1179 files belonging to 1 classes.
Using 944 files for training.
Found 1179 files belonging to 1 classes.
Using 235 files for validation.
```

[ ]:
```
# visualize a few random butterfly
plt.figure(figsize=(12, 28))

for images, labels in butterfly_generator.take(1):
  for i in range(len(images)):
      ax = plt.subplot(8, 4, i + 1)
      plt.imshow(images[i].numpy().astype("uint8"))
      plt.title(butterfly_names[labels[i]])
      plt.axis("off")
```

VICEROY

ULYSES

GLITTERING SAPPHIRE

ORCHARD SWALLOW

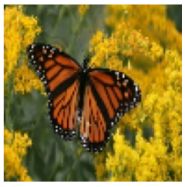
PURPLE HAIRSTREAK

ULYSES

PURPLE HAIRSTREAK

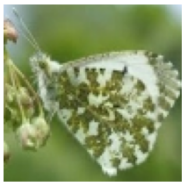
GLITTERING SAPPHIRE

MONARCH



EASTERN DAPPLE WHITE

EASTERN DAPPLE WHITE



VICEROY

VICEROY

EASTERN DAPPLE WHITE

VICEROY

YELLOW SWALLOW TAIL

YELLOW SWALLOW TAIL

YELLOW SWALLOW TAIL

BLUE MORPHO

VICEROY

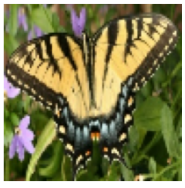
BLUE MORPHO

ULYSES

EASTERN DAPPLE WHITE

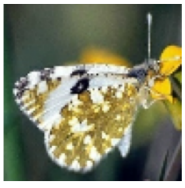
MONARCH

YELLOW SWALLOW TAIL



BLUE MORPHO

EASTERN DAPPLE WHITE



ADONIS

MONARCH

PURPLE HAIRSTREAK

ORCHARD SWALLOW

VICEROY

[ ]:
```
ENCODING_SIZE = 8

# build a VAE based on AE structure above
class Sampling(keras.layers.Layer):
  def call(self, inputs):
    mean, log_var = inputs
    return K.random_normal(tf.shape(log_var)) * K.exp(log_var /2) + mean

inputs = keras.layers.Input(shape = [IMG_WIDTH, IMG_HEIGHT, 3])
z = keras.layers.Conv2D(64, input_shape = [IMG_WIDTH, IMG_HEIGHT, 3],␣
 ↪kernel_size = 3,
                        strides = 1, padding = 'same', activation = 'selu',
                        kernel_initializer='lecun_normal')(inputs)
z = keras.layers.Conv2D(128, kernel_size = 3, strides = 2,
                        padding = 'same', activation = 'selu',
                        kernel_initializer='lecun_normal')(z)
z = keras.layers.Conv2D(256, kernel_size = 3, strides = 2,
                        padding = 'same', activation = 'selu',
                        kernel_initializer='lecun_normal')(z)
z = keras.layers.Conv2D(512, kernel_size = 3, strides = 2,
                        padding = 'same', activation = 'selu',
                        kernel_initializer='lecun_normal')(z)
z = keras.layers.Flatten()(z)
codings_mean = keras.layers.Dense(ENCODING_SIZE)(z)
codings_log_var = keras.layers.Dense(ENCODING_SIZE)(z)
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = keras.Model(inputs = [inputs], outputs = [codings_mean,␣
 ↪codings_log_var, codings])

variational_encoder.summary()

decoder_inputs = keras.layers.Input(shape = [ENCODING_SIZE])
x = keras.layers.Dense(512 * 12 * 12, activation = "selu", input_shape =␣
 ↪[ENCODING_SIZE])(decoder_inputs)
x = keras.layers.Reshape([12, 12, 512])(x)
x = keras.layers.Conv2DTranspose(filters = 512, kernel_size = 3, strides = 1,
                                 padding = "same", activation = "selu")(x)
x = keras.layers.Conv2DTranspose(filters = 256, kernel_size = 3, strides = 2,
                                 padding = "same", activation = "selu")(x)
x = keras.layers.Conv2DTranspose(filters = 128, kernel_size = 3, strides = 2,
                                 padding = "same", activation = "selu")(x)
x = keras.layers.Conv2DTranspose(filters = 64, kernel_size = 3, strides = 2,
                                 padding = "same", activation = "selu")(x)
```

```
outputs = keras.layers.Conv2DTranspose(filters = 3, kernel_size = 3, strides =␣
 ↪1,
                                       padding = 'same', activation =␣
 ↪'sigmoid')(x)
variational_decoder = keras.Model(inputs = [decoder_inputs], outputs =␣
 ↪[outputs])

variational_decoder.summary()

_, _, codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
vae = keras.Model(inputs = [inputs], outputs = [reconstructions])

vae.summary()

# add a loss function
latent_loss = -0.5 * K.sum(1 + codings_log_var - K.exp(codings_log_var) - K.
 ↪square(codings_mean),
                           axis = -1)
vae.add_loss(K.mean(latent_loss) / 27648.)
```

```
Model: "model"
__________________________________________________________________________________
__________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================
==================
 input_1 (InputLayer)        [(None, 96, 96, 3)]       0         []

 conv2d (Conv2D)             (None, 96, 96, 64)        1792
['input_1[0][0]']

 conv2d_1 (Conv2D)           (None, 48, 48, 128)       73856
['conv2d[0][0]']

 conv2d_2 (Conv2D)           (None, 24, 24, 256)       295168
['conv2d_1[0][0]']

 conv2d_3 (Conv2D)           (None, 12, 12, 512)       1180160
['conv2d_2[0][0]']

 flatten (Flatten)           (None, 73728)             0
['conv2d_3[0][0]']

 dense (Dense)               (None, 8)                 589832
['flatten[0][0]']
```

```
 dense_1 (Dense)             (None, 8)                    589832
['flatten[0][0]']

 sampling (Sampling)         (None, 8)                    0
['dense[0][0]',
'dense_1[0][0]']

==================================================================================
==================
Total params: 2730640 (10.42 MB)
Trainable params: 2730640 (10.42 MB)
----------------------------------------------------------------------------------
------------------
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================
==================
 input_1 (InputLayer)        [(None, 96, 96, 3)]          0         []

 conv2d (Conv2D)             (None, 96, 96, 64)           1792
['input_1[0][0]']

 conv2d_1 (Conv2D)           (None, 48, 48, 128)          73856
['conv2d[0][0]']

 conv2d_2 (Conv2D)           (None, 24, 24, 256)          295168
['conv2d_1[0][0]']

 conv2d_3 (Conv2D)           (None, 12, 12, 512)          1180160
['conv2d_2[0][0]']

 flatten (Flatten)           (None, 73728)                0
['conv2d_3[0][0]']

 dense (Dense)               (None, 8)                    589832
['flatten[0][0]']

 dense_1 (Dense)             (None, 8)                    589832
['flatten[0][0]']

 sampling (Sampling)         (None, 8)                    0
['dense[0][0]',
'dense_1[0][0]']

==================================================================================
==================
Total params: 2730640 (10.42 MB)
Trainable params: 2730640 (10.42 MB)
Non-trainable params: 0 (0.00 Byte)
```

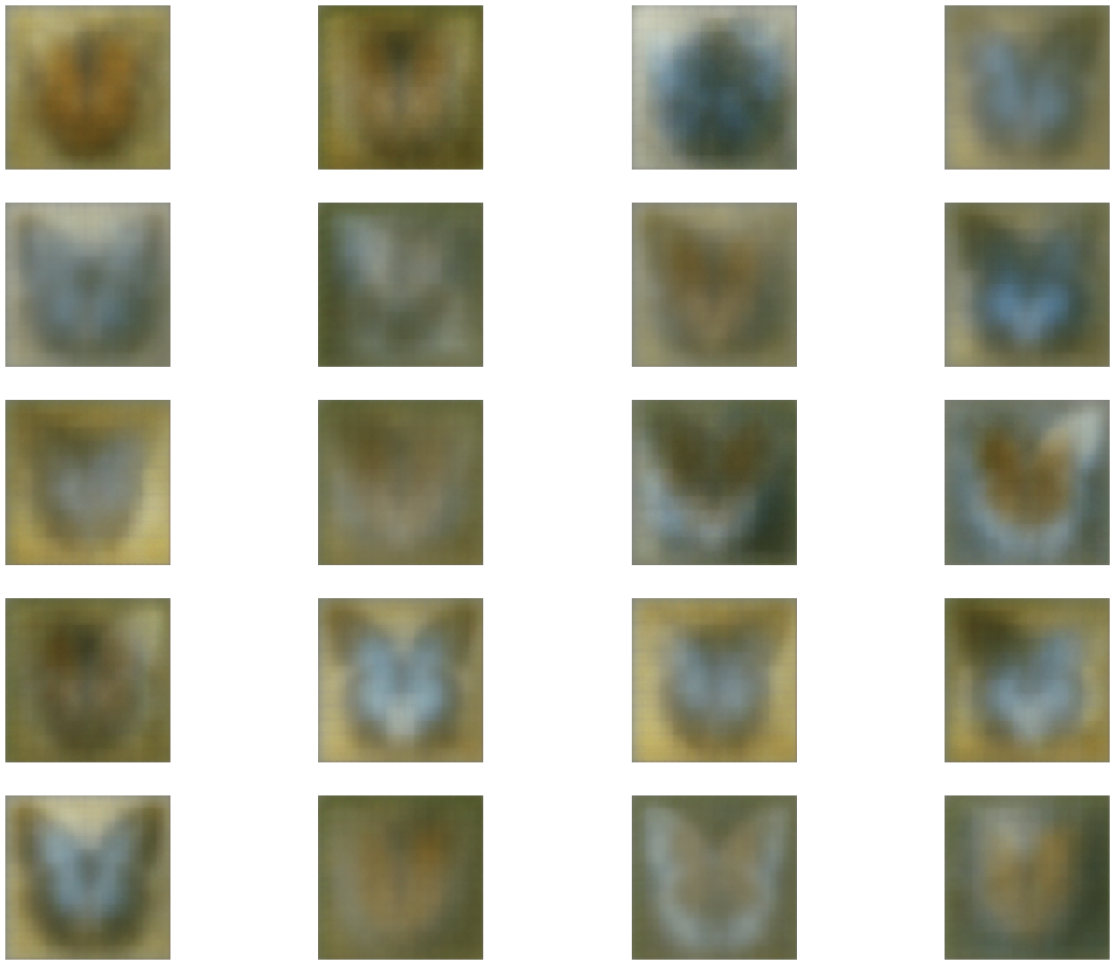
```
_________________________________________________________________
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 8)]               0

 dense_2 (Dense)             (None, 73728)             663552

 reshape (Reshape)           (None, 12, 12, 512)       0

 conv2d_transpose (Conv2DTr  (None, 12, 12, 512)       2359808
 anspose)

 conv2d_transpose_1 (Conv2D  (None, 24, 24, 256)       1179904
 Transpose)

 conv2d_transpose_2 (Conv2D  (None, 48, 48, 128)       295040
 Transpose)

 conv2d_transpose_3 (Conv2D  (None, 96, 96, 64)        73792
 Transpose)

 conv2d_transpose_4 (Conv2D  (None, 96, 96, 3)         1731
 Transpose)

=================================================================
Total params: 4573827 (17.45 MB)
Trainable params: 4573827 (17.45 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 96, 96, 3)]       0

 model (Functional)          [(None, 8),               2730640
                              (None, 8),
                              (None, 8)]

 model_1 (Functional)        (None, 96, 96, 3)         4573827

=================================================================
Total params: 7304467 (27.86 MB)
Trainable params: 7304467 (27.86 MB)
Non-trainable params: 0 (0.00 Byte)
```

[ ]:
```
# add learning rate scheduling
def exponential_decay_fn(epoch):
  return 0.0001 * 0.1 ** (epoch / 10)

lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)

# add early stopping
early_stopping_cb = keras.callbacks.EarlyStopping(patience = 5,␣
 ↪restore_best_weights = True)

# clear the session for a clean run
keras.backend.clear_session()
tf.random.set_seed(42)

vae.compile(loss = 'mean_squared_error', optimizer = keras.optimizers.Nadam())
history = vae.fit(X_train, validation_data = X_valid,  epochs = 20, \
                  callbacks = [lr_scheduler, early_stopping_cb])
```

```
Epoch 1/20
30/30 [==============================] - 55s 2s/step - loss: 0.0773 - val_loss:
0.0675 - lr: 1.0000e-04
Epoch 2/20
30/30 [==============================] - 52s 2s/step - loss: 0.0615 - val_loss:
0.0580 - lr: 7.9433e-05
Epoch 3/20
30/30 [==============================] - 50s 2s/step - loss: 0.0539 - val_loss:
0.0526 - lr: 6.3096e-05
Epoch 4/20
30/30 [==============================] - 50s 2s/step - loss: 0.0502 - val_loss:
0.0494 - lr: 5.0119e-05
Epoch 5/20
30/30 [==============================] - 51s 2s/step - loss: 0.0477 - val_loss:
0.0475 - lr: 3.9811e-05
Epoch 6/20
30/30 [==============================] - 52s 2s/step - loss: 0.0458 - val_loss:
0.0462 - lr: 3.1623e-05
Epoch 7/20
30/30 [==============================] - 53s 2s/step - loss: 0.0446 - val_loss:
0.0457 - lr: 2.5119e-05
Epoch 8/20
30/30 [==============================] - 53s 2s/step - loss: 0.0441 - val_loss:
0.0455 - lr: 1.9953e-05
Epoch 9/20
30/30 [==============================] - 52s 2s/step - loss: 0.0437 - val_loss:
0.0453 - lr: 1.5849e-05
Epoch 10/20
```

```
30/30 [==============================] - 50s 2s/step - loss: 0.0435 - val_loss:
0.0451 - lr: 1.2589e-05
Epoch 11/20
30/30 [==============================] - 52s 2s/step - loss: 0.0433 - val_loss:
0.0450 - lr: 1.0000e-05
Epoch 12/20
30/30 [==============================] - 54s 2s/step - loss: 0.0432 - val_loss:
0.0450 - lr: 7.9433e-06
Epoch 13/20
30/30 [==============================] - 56s 2s/step - loss: 0.0431 - val_loss:
0.0449 - lr: 6.3096e-06
Epoch 14/20
30/30 [==============================] - 50s 2s/step - loss: 0.0430 - val_loss:
0.0449 - lr: 5.0119e-06
Epoch 15/20
30/30 [==============================] - 50s 2s/step - loss: 0.0429 - val_loss:
0.0448 - lr: 3.9811e-06
Epoch 16/20
30/30 [==============================] - 50s 2s/step - loss: 0.0429 - val_loss:
0.0449 - lr: 3.1623e-06
Epoch 17/20
30/30 [==============================] - 50s 2s/step - loss: 0.0428 - val_loss:
0.0448 - lr: 2.5119e-06
Epoch 18/20
30/30 [==============================] - 51s 2s/step - loss: 0.0428 - val_loss:
0.0448 - lr: 1.9953e-06
Epoch 19/20
30/30 [==============================] - 51s 2s/step - loss: 0.0428 - val_loss:
0.0448 - lr: 1.5849e-06
Epoch 20/20
30/30 [==============================] - 55s 2s/step - loss: 0.0428 - val_loss:
0.0449 - lr: 1.2589e-06
```

[ ]:
```
# visualizing few butterfly images based on random sampling of means
random_codings = tf.random.normal(shape = [20, ENCODING_SIZE])
images = variational_decoder(random_codings).numpy()

plt.figure(figsize=(15, 12))

for i, image in enumerate(images):
  ax = plt.subplot(5, 4, i + 1)
  plt.imshow(image)
  plt.axis("off")
```



[ ]:
```
# set up a dictionary to store a few images of each butterfly
butterfly_means = {name:{'count':0, 'means':[0 for _ in range(ENCODING_SIZE)]}␣
 ↪for name in butterfly_names}

# hold onto a set of examples from each butterfly type
butterfly_examples = {}

# for each butterfly image, run it through the encoder and extract the encodings
# getting the mean of the encodings for each butterfly type
# using the generator created
for images, labels in butterfly_generator:
  means, gammas, coding = variational_encoder.predict(images / 255)
  for idx, mean in enumerate(means):
    butterfly_name = butterfly_names[labels[idx]]
    butterfly_means[butterfly_name]['count'] += 1

    for jdx, element in enumerate(mean):
```

```
      butterfly_means[butterfly_name]['means'][jdx] += element

    if butterfly_name not in butterfly_examples.keys():
      butterfly_examples[butterfly_name] = []
    if len(butterfly_examples[butterfly_name]) < 3:
      butterfly_examples[butterfly_name].append(images[idx] / 255)
    else:
      if random.random() < 0.5:
        butterfly_examples[butterfly_name][random.randint(0,2)] = images[idx] /␣
 ↪255

for k,v in butterfly_means.items():
  butterfly_means[k]['means'] = [m / butterfly_means[k]['count'] for m in␣
 ↪butterfly_means[k]['means']]
```

```
1/1 [==============================] - 0s 179ms/step
1/1 [==============================] - 0s 179ms/step
1/1 [==============================] - 0s 132ms/step
1/1 [==============================] - 0s 116ms/step
1/1 [==============================] - 0s 125ms/step
1/1 [==============================] - 0s 106ms/step
1/1 [==============================] - 0s 131ms/step
1/1 [==============================] - 0s 122ms/step
1/1 [==============================] - 0s 112ms/step
1/1 [==============================] - 0s 145ms/step
1/1 [==============================] - 0s 127ms/step
1/1 [==============================] - 0s 141ms/step
1/1 [==============================] - 0s 116ms/step
1/1 [==============================] - 0s 114ms/step
1/1 [==============================] - 0s 140ms/step
1/1 [==============================] - 0s 144ms/step
1/1 [==============================] - 0s 120ms/step
1/1 [==============================] - 0s 114ms/step
1/1 [==============================] - 0s 115ms/step
1/1 [==============================] - 0s 128ms/step
1/1 [==============================] - 0s 117ms/step
1/1 [==============================] - 0s 128ms/step
1/1 [==============================] - 0s 113ms/step
1/1 [==============================] - 0s 132ms/step
1/1 [==============================] - 0s 131ms/step
1/1 [==============================] - 0s 134ms/step
1/1 [==============================] - 0s 149ms/step
1/1 [==============================] - 0s 121ms/step
1/1 [==============================] - 0s 114ms/step
1/1 [==============================] - 0s 120ms/step
1/1 [==============================] - 0s 113ms/step
1/1 [==============================] - 0s 112ms/step
1/1 [==============================] - 0s 124ms/step
```

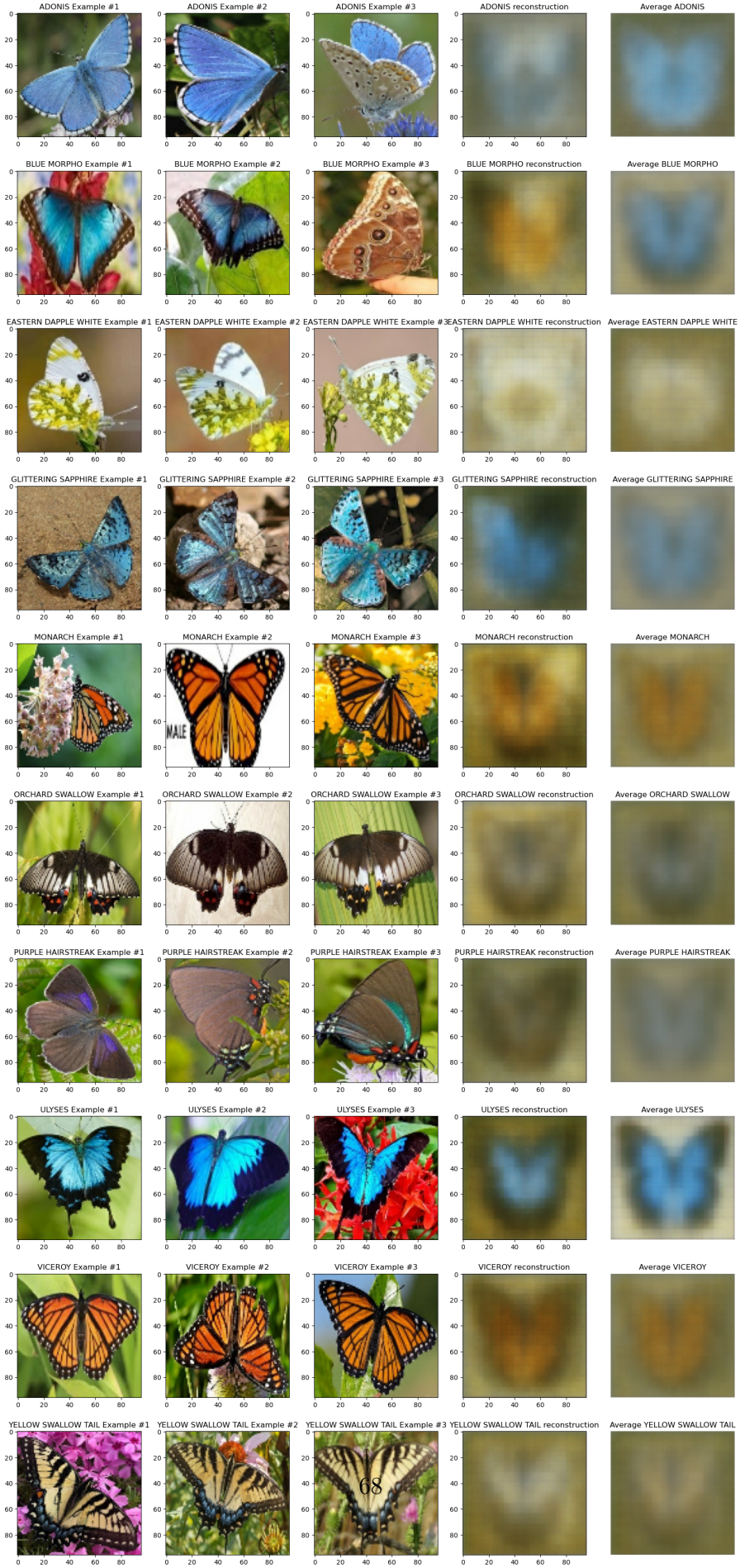
```
1/1 [==============================] - 0s 131ms/step
1/1 [==============================] - 0s 115ms/step
1/1 [==============================] - 0s 112ms/step
1/1 [==============================] - 0s 110ms/step
1/1 [==============================] - 0s 140ms/step
```

[ ]:
```
# process mean vectors through decoder to generate
for i, (k, v) in enumerate(butterfly_means.items()):
  butterfly_means[k]['average_butterfly']= variational_decoder.
 ↪predict([v['means']])[0]
```

```
1/1 [==============================] - 0s 113ms/step
1/1 [==============================] - 0s 113ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 45ms/step
```

[ ]:
```
# compare examples, an example run through VAE
# and average butterfly
plt.figure(figsize=(20, 70))

for i, (k, v) in enumerate(butterfly_means.items()):
  original_example = butterfly_examples[k][2] # just process the third one
  processed_example = vae.predict(original_example[None, ...])
  average_butterfly = butterfly_means[k]['average_butterfly']

  ax = plt.subplot(16, 5, i * 5 + 1)
  plt.imshow(butterfly_examples[k][0])
  plt.title(k + ' Example #1')
  ax = plt.subplot(16, 5, i * 5 + 2)
  plt.imshow(butterfly_examples[k][1])
  plt.title(k + ' Example #2')
  ax = plt.subplot(16, 5, i * 5 + 3)
  plt.imshow(butterfly_examples[k][2])
  plt.title(k + ' Example #3')
  ax = plt.subplot(16, 5, i * 5 + 4)
  plt.imshow(processed_example[0])
  plt.title(k + ' reconstruction')
  ax = plt.subplot(16, 5, i * 5 + 5)
  plt.imshow(average_butterfly)
  plt.title('Average ' + k)
  plt.axis("off")
```

```
1/1 [==============================] - 0s 176ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 31ms/step
```


ADONIS Example #1
ADONIS Example #2
ADONIS Example #3
ADONIS reconstruction
Average ADONIS
BLUE MORPHO Example #1
BLUE MORPHO Example #2
BLUE MORPHO Example #3
BLUE MORPHO reconstruction
Average BLUE MORPHO
EASTERN DAPPLE WHITE Example #1
EASTERN DAPPLE WHITE Example #2
EASTERN DAPPLE WHITE Example #3
EASTERN DAPPLE WHITE reconstruction
Average EASTERN DAPPLE WHITE
GLITTERING SAPPHIRE Example #1
GLITTERING SAPPHIRE Example #2
GLITTERING SAPPHIRE Example #3
GLITTERING SAPPHIRE reconstruction
Average GLITTERING SAPPHIRE
MONARCH Example #1
MONARCH Example #2
MONARCH Example #3
MONARCH reconstruction
Average MONARCH
ORCHARD SWALLOW Example #1
ORCHARD SWALLOW Example #2
ORCHARD SWALLOW Example #3
ORCHARD SWALLOW reconstruction
Average ORCHARD SWALLOW
PURPLE HAIRSTREAK Example #1
PURPLE HAIRSTREAK Example #2
PURPLE HAIRSTREAK Example #3
PURPLE HAIRSTREAK reconstruction
Average PURPLE HAIRSTREAK
ULYSES Example #1
ULYSES Example #2
ULYSES Example #3
ULYSES reconstruction
Average ULYSES
VICEROY Example #1
VICEROY Example #2
VICEROY Example #3
VICEROY reconstruction
Average VICEROY
YELLOW SWALLOW TAIL Example #1
YELLOW SWALLOW TAIL Example #2
YELLOW SWALLOW TAIL Example #3
YELLOW SWALLOW TAIL reconstruction
Average YELLOW SWALLOW TAIL

Analysis: In this part of the activity, it shows how VAE work on the dataset chosen which us the butterfly dataset. Creation of encoder and decoder were shown in order for the VAE to fully demonstrated. Also, the output shows how VAE generate the original data as the epochs increase. The plot image shows the simulated images created by the VAE algorithm created which is also a basis for the model's performance in generating data.

3 Conclusion

In this activity we were tasked to perform autoencoders, GAN and VAE in MNIST dataset as well as the dataset that we have chosen. Both GAN and VAE are usually used to generate synthetic data which will add to the dataset for a better training for the model in purpose of image classification and other applications. These algorithms are useful in generating data which can be of used to enhance the performance of a model. The tasks included in this activity will show the demonstration of autoencoder, VAN and GAE which is helpful in visualizing the usage of the mentioned algorithms. These three frameworks is a useful tool in generating data as well as generating its simulated form which will test the performance of the model being used. The supplementary part allows us to use dataset we have chosen which is a good example for visualizing the how autoencoders, GAN and VAE works.

Link: https://colab.research.google.com/drive/1Vbfcqnan099A3wBPmKlU1IEBQhPNv0eM?usp=sharing

4 References

Variational Autoencoders and GANs

Harvard University

Fall 2020

Instructors: Mark Glickman, Pavlos Protopapas, and Chris Tanner

Lab Instructors: Chris Tanner and Eleni Angelaki Kaxiras

Content: Srivatsan Srinivasan, Pavlos Protopapas, Chris Tanner

Link: https://harvard-iacs.github.io/2020-CS109B/labs/lab10/notebook/