Week 3: Variational Autoencoders on Anime Faces

For this exercise, you will train a Variational Autoencoder (VAE) using the anime faces dataset by MckInsey666.

You will train the model using the techniques discussed in class. At the end, you should save your model and download it from Colab so that it can be submitted to the autograder for grading.

Important: *This colab notebook has read-only access so you won't be able to save your changes. If you want to save your work periodically, please click* `File -> Save a Copy in Drive` *to create a copy in your account, then work from there.*

Imports

```
import tensorflow as tf
import tensorflow_datasets as tfds

import matplotlib.pyplot as plt
import numpy as np

import os
import zipfile
import urllib.request
import random
from IPython import display
```

Parameters

```
# set a random seed
np.random.seed(51)

# parameters for building the model and training
BATCH_SIZE=2000
LATENT_DIM=512
IMAGE_SIZE=64
```

Download the Dataset

You will download the Anime Faces dataset and save it to a local directory.

```
# make the data directory
try:
  os.mkdir('/tmp/anime')
except OSError:
  pass

# download the zipped dataset to the data directory
```

```
data_url = "https://storage.googleapis.com/laurencemoroney-blog.appspot.com/Resources/anime-faces.zi
data_file_name = "animefaces.zip"
download_dir = '/tmp/anime/'
urllib.request.urlretrieve(data_url, data_file_name)

# extract the zip file
zip_ref = zipfile.ZipFile(data_file_name, 'r')
zip_ref.extractall(download_dir)
zip_ref.close()
```

Prepare the Dataset

Next is preparing the data for training and validation. We've provided you some utilities below.

```
# Data Preparation Utilities

def get_dataset_slice_paths(image_dir):
  '''returns a list of paths to the image files'''
  image_file_list = os.listdir(image_dir)
  image_paths = [os.path.join(image_dir, fname) for fname in image_file_list]

  return image_paths


def map_image(image_filename):
  '''preprocesses the images'''
  img_raw = tf.io.read_file(image_filename)
  image = tf.image.decode_jpeg(img_raw)

  image = tf.cast(image, dtype=tf.float32)
  image = tf.image.resize(image, (IMAGE_SIZE, IMAGE_SIZE))
  image = image / 255.0
  image = tf.reshape(image, shape=(IMAGE_SIZE, IMAGE_SIZE, 3,))

  return image
```

You will use the functions above to generate the train and validation sets.

```
# get the list containing the image paths
paths = get_dataset_slice_paths("/tmp/anime/images/")

# shuffle the paths
random.shuffle(paths)

# split the paths list into to training (80%) and validation sets(20%).
paths_len = len(paths)
train_paths_len = int(paths_len * 0.8)

train_paths = paths[:train_paths_len]
val_paths = paths[train_paths_len:]

# load the training image paths into tensors, create batches and shuffle
training_dataset = tf.data.Dataset.from_tensor_slices((train_paths))
training_dataset = training_dataset.map(map_image)
```

```
training_dataset = training_dataset.shuffle(1000).batch(BATCH_SIZE)

# load the validation image paths into tensors and create batches
validation_dataset = tf.data.Dataset.from_tensor_slices((val_paths))
validation_dataset = validation_dataset.map(map_image)
validation_dataset = validation_dataset.batch(BATCH_SIZE)


print(f'number of batches in the training set: {len(training_dataset)}')
print(f'number of batches in the validation set: {len(validation_dataset)}')
```

```
number of batches in the training set: 26
number of batches in the validation set: 7
```

Display Utilities

We've also provided some utilities to help in visualizing the data.

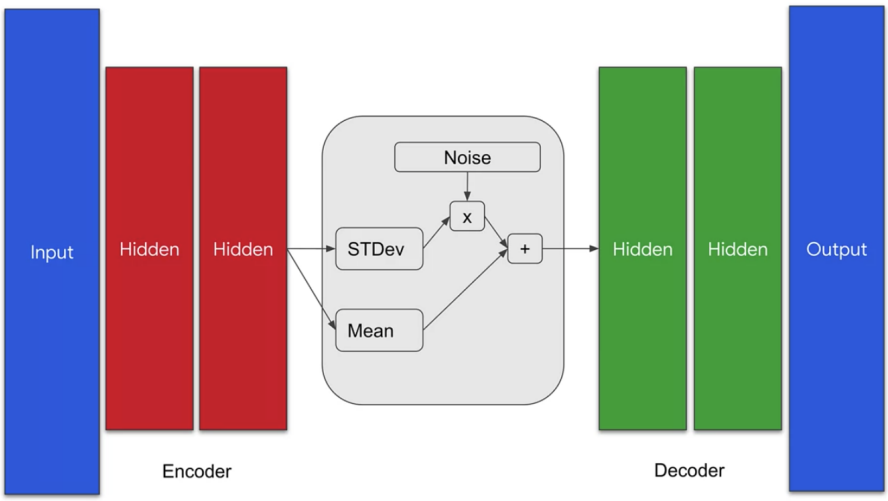
```
def display_faces(dataset, size=9):
  '''Takes a sample from a dataset batch and plots it in a grid.'''
  dataset = dataset.unbatch().take(size)
  n_cols = 3
  n_rows = size//n_cols + 1
  plt.figure(figsize=(5, 5))
  i = 0
  for image in dataset:
    i += 1
    disp_img = np.reshape(image, (64,64,3))
    plt.subplot(n_rows, n_cols, i)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(disp_img)


def display_one_row(disp_images, offset, shape=(28, 28)):
  '''Displays a row of images.'''
  for idx, image in enumerate(disp_images):
    plt.subplot(3, 10, offset + idx + 1)
    plt.xticks([])
    plt.yticks([])
    image = np.reshape(image, shape)
    plt.imshow(image)


def display_results(disp_input_images, disp_predicted):
  '''Displays input and predicted images.'''
  plt.figure(figsize=(15, 5))
  display_one_row(disp_input_images, 0, shape=(IMAGE_SIZE,IMAGE_SIZE,3))
  display_one_row(disp_predicted, 20, shape=(IMAGE_SIZE,IMAGE_SIZE,3))
```

Let's see some of the anime faces from the validation dataset.

```
display_faces(validation_dataset, size=12)
```

Build the Model

You will be building your VAE in the following sections. Recall that this will follow and encoder-decoder architecture and can be summarized by the figure below.



Sampling Class

You will start with the custom layer to provide the Gaussian noise input along with the mean (mu) and standard deviation (sigma) of the encoder's output. Recall the equation to combine these:

$$z = \mu + e^{0.5\sigma} * \epsilon$$

where μ = mean, σ = standard deviation, and ϵ = random sample

```
class Sampling(tf.keras.layers.Layer):
  def call(self, inputs):
    """Generates a random sample and combines with the encoder output

    Args:
      inputs -- output tensor from the encoder

    Returns:
      `inputs` tensors combined with a random sample
    """
    ### START CODE HERE ###
    mu, sigma = inputs
    batch = tf.shape(mu)[0]
```

```
    dim = tf.shape(mu)[1]
    epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
    z = mu + tf.exp(0.5 * sigma) * epsilon
    ### END CODE HERE ###
    return  z
```

Encoder Layers

Next, please use the Functional API to stack the encoder layers and output `mu`, `sigma` and the shape of the features before flattening. We expect you to use 3 convolutional layers (instead of 2 in the ungraded lab) but feel free to revise as you see fit. Another hint is to use `1024` units in the Dense layer before you get mu and sigma (we used `20` for it in the ungraded lab).

Note: If you did Week 4 before Week 3, please do not use LeakyReLU activations yet for this particular assignment. The grader for Week 3 does not support LeakyReLU yet. This will be updated but for now, you can use `relu` and `sigmoid` just like in the ungraded lab.

```
def encoder_layers(inputs, latent_dim):
  """Defines the encoder's layers.
  Args:
    inputs -- batch from the dataset
    latent_dim -- dimensionality of the latent space

  Returns:
    mu -- learned mean
    sigma -- learned standard deviation
    batch_3.shape -- shape of the features before flattening
  """
  ### START CODE HERE ###
  # add the Conv2D layers followed by BatchNormalization
  x = tf.keras.layers.Conv2D(filters=32, kernel_size=3, strides=2, padding="same", activation='relu
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Conv2D(filters=64, kernel_size=3, strides=2, padding='same', activation='relu
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Conv2D(filters=128, kernel_size=3, strides=2, padding='same', activation='relu

  # assign to a different variable so you can extract the shape later
  batch_3 = tf.keras.layers.BatchNormalization()(x)

  # flatten the features and feed into the Dense network
  x = tf.keras.layers.Flatten(name="encode_flatten")(batch_3)

  # we arbitrarily used 20 units here but feel free to change and see what results you get
  x = tf.keras.layers.Dense(1024, activation='relu', name="encode_dense")(x)
  x = tf.keras.layers.BatchNormalization()(x)

  # add output Dense networks for mu and sigma, units equal to the declared latent_dim.
  mu = tf.keras.layers.Dense(latent_dim, name='latent_mu')(x)
  sigma = tf.keras.layers.Dense(latent_dim, name ='latent_sigma')(x)

  ### END CODE HERE ###

  # revise `batch_3.shape` here if you opted not to use 3 Conv2D layers
  return mu, sigma, batch_3.shape
```

Encoder Model

You will feed the output from the above function to the `Sampling layer` you defined earlier. That will have the latent representations that can be fed to the decoder network later. Please complete the function below to build the encoder network with the `Sampling` layer.

```
def encoder_model(latent_dim, input_shape):
  """Defines the encoder model with the Sampling layer
  Args:
    latent_dim -- dimensionality of the latent space
    input_shape -- shape of the dataset batch

  Returns:
    model -- the encoder model
    conv_shape -- shape of the features before flattening
  """
  ### START CODE HERE ###
  inputs = tf.keras.layers.Input(shape=input_shape)
  mu, sigma, conv_shape = encoder_layers(inputs, latent_dim=LATENT_DIM)
  z = Sampling()((mu, sigma))
  model = tf.keras.Model(inputs, outputs=[mu, sigma, z])
  ### END CODE HERE ###
  model.summary()
  return model, conv_shape
```

Decoder Layers

Next, you will define the decoder layers. This will expand the latent representations back to the original image dimensions. After training your VAE model, you can use this decoder model to generate new data by feeding random inputs.

```
def decoder_layers(inputs, conv_shape):
  """Defines the decoder layers.
  Args:
    inputs -- output of the encoder
    conv_shape -- shape of the features before flattening

  Returns:
    tensor containing the decoded output
  """
  ### START CODE HERE ###
  # feed to a Dense network with units computed from the conv_shape dimensions
  units = conv_shape[1] * conv_shape[2] * conv_shape[3]
  x = tf.keras.layers.Dense(units, activation = 'relu', name="decode_dense1")(inputs)
  x = tf.keras.layers.BatchNormalization()(x)

  # reshape output using the conv_shape dimensions
  x = tf.keras.layers.Reshape((conv_shape[1], conv_shape[2], conv_shape[3]), name="decode_reshape")(

  # upsample the features back to the original dimensions
  x = tf.keras.layers.Conv2DTranspose(filters=128, kernel_size=3, strides=2, padding='same', activat
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Conv2DTranspose(filters=64, kernel_size=3, strides=2, padding='same', activati
  x = tf.keras.layers.BatchNormalization()(x)
```

```
  x = tf.keras.layers.Conv2DTranspose(filters=32, kernel_size=3, strides=2, padding='same', activati
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Conv2DTranspose(filters=3, kernel_size=3, strides=1, padding='same', activatio

  ### END CODE HERE ###
  return x
```

Decoder Model

Please complete the function below to output the decoder model.

```
def decoder_model(latent_dim, conv_shape):
  """Defines the decoder model.
  Args:
    latent_dim -- dimensionality of the latent space
    conv_shape -- shape of the features before flattening

  Returns:
    model -- the decoder model
  """
  ### START CODE HERE ###
  inputs = tf.keras.layers.Input(shape=(latent_dim,))
  outputs = decoder_layers(inputs, conv_shape)
  model = tf.keras.Model(inputs, outputs)
  ### END CODE HERE ###
  model.summary()
  return model
```

Kullback–Leibler Divergence

Next, you will define the function to compute the Kullback–Leibler Divergence loss. This will be used to improve the generative capability of the model. This code is already given.

```
def kl_reconstruction_loss(inputs, outputs, mu, sigma):
  """ Computes the Kullback-Leibler Divergence (KLD)
  Args:
    inputs -- batch from the dataset
    outputs -- output of the Sampling layer
    mu -- mean
    sigma -- standard deviation

  Returns:
    KLD loss
  """
  kl_loss = 1 + sigma - tf.square(mu) - tf.math.exp(sigma)
  return tf.reduce_mean(kl_loss) * -0.5
```

Putting it all together

Please define the whole VAE model. Remember to use `model.add_loss()` to add the KL reconstruction loss. This will be accessed and added to the loss later in the training loop.

```
def vae_model(encoder, decoder, input_shape):
```

```
def vae_model(encoder, decoder, input_shape):
  """Defines the VAE model
  Args:
    encoder -- the encoder model
    decoder -- the decoder model
    input_shape -- shape of the dataset batch

  Returns:
    the complete VAE model
  """
  ### START CODE HERE ###
  # set the inputs
  inputs = tf.keras.layers.Input(shape=input_shape)

  # get mu, sigma, and z from the encoder output
  mu, sigma, z = encoder(inputs)

  # get reconstructed output from the decoder
  reconstructed = decoder(z)

  # define the inputs and outputs of the VAE
  model = tf.keras.Model(inputs=inputs, outputs=reconstructed)

  # add the KL loss
  loss = kl_reconstruction_loss(inputs, z, mu, sigma)
  model.add_loss(loss)

  ### END CODE HERE ###
  return model
```

Next, please define a helper function to return the encoder, decoder, and vae models you just defined.

```
def get_models(input_shape, latent_dim):
  """Returns the encoder, decoder, and vae models"""
  ### START CODE HERE ###
  encoder, conv_shape = encoder_model(latent_dim=latent_dim, input_shape=input_shape)
  decoder = decoder_model(latent_dim=latent_dim, conv_shape=conv_shape)
  vae = vae_model(encoder, decoder, input_shape=input_shape)

  ### END CODE HERE ###
  return encoder, decoder, vae
```

Let's use the function above to get the models we need in the training loop.

```
encoder, decoder, vae = get_models(input_shape=(64,64,3,), latent_dim=LATENT_DIM)
```

```
encode_conv1 (Conv2D)           (None, 32, 32, 32)   896         input_4[0][0]
batch_normalization_8 (BatchNor (None, 32, 32, 32)   128         encode_conv1[0][0]
encode_conv2 (Conv2D)           (None, 16, 16, 64)   18496       batch_normalization_8[0][0
batch_normalization_9 (BatchNor (None, 16, 16, 64)   256         encode_conv2[0][0]
encode_conv3 (Conv2D)           (None, 8, 8, 128)    73856       batch_normalization_9[0][0
batch_normalization_10 (BatchNo (None, 8, 8, 128)    512         encode_conv3[0][0]
```

```
encode_flatten (Flatten)        (None, 8192)         0           batch_normalization_10[0][
__________________________________________________________________________________________
encode_dense (Dense)            (None, 1024)         8389632     encode_flatten[0][0]
__________________________________________________________________________________________
batch_normalization_11 (BatchNo (None, 1024)         4096        encode_dense[0][0]
__________________________________________________________________________________________
latent_mu (Dense)               (None, 512)          524800      batch_normalization_11[0][
__________________________________________________________________________________________
latent_sigma (Dense)            (None, 512)          524800      batch_normalization_11[0][
__________________________________________________________________________________________
sampling_1 (Sampling)           (None, 512)          0           latent_mu[0][0]
                                                                 latent_sigma[0][0]
==========================================================================================
Total params: 9,537,472
Trainable params: 9,534,976
Non-trainable params: 2,496
__________________________________________________________________________________________
Model: "model_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         [(None, 512)]             0
_________________________________________________________________
decode_dense1 (Dense)        (None, 8192)              4202496
_________________________________________________________________
batch_normalization_12 (Batc (None, 8192)              32768
_________________________________________________________________
decode_reshape (Reshape)     (None, 8, 8, 128)         0
_________________________________________________________________
decode_conv2d_1 (Conv2DTrans (None, 16, 16, 128)       147584
_________________________________________________________________
batch_normalization_13 (Batc (None, 16, 16, 128)       512
_________________________________________________________________
decode_conv2d_2 (Conv2DTrans (None, 32, 32, 64)        73792
_________________________________________________________________
batch_normalization_14 (Batc (None, 32, 32, 64)        256
_________________________________________________________________
decode_conv2d_3 (Conv2DTrans (None, 64, 64, 32)        18464
_________________________________________________________________
batch_normalization_15 (Batc (None, 64, 64, 32)        128
_________________________________________________________________
decode_final (Conv2DTranspos (None, 64, 64, 3)         867
=================================================================
Total params: 4,476,867
Trainable params: 4,460,035
Non-trainable params: 16,832
```

Train the Model

You will now configure the model for training. We defined some losses, the optimizer, and the loss metric below but you can experiment with others if you like.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.002)
loss_metric = tf.keras.metrics.Mean()
mse_loss = tf.keras.losses.MeanSquaredError()
bce_loss = tf.keras.losses.BinaryCrossentropy()
```

You will generate 16 images in a 4x4 grid to show progress of image generation. We've defined a utility function for that below.

```
def generate_and_save_images(model, epoch, step, test_input):
  """Helper function to plot our 16 images

  Args:

  model -- the decoder model
  epoch -- current epoch number during training
  step -- current step number during training
  test_input -- random tensor with shape (16, LATENT_DIM)
  """
  predictions = model.predict(test_input)

  fig = plt.figure(figsize=(4,4))

  for i in range(predictions.shape[0]):
      plt.subplot(4, 4, i+1)
      img = predictions[i, :, :, :] * 255
      img = img.astype('int32')
      plt.imshow(img)
      plt.axis('off')

  # tight_layout minimizes the overlap between 2 sub-plots
  fig.suptitle("epoch: {}, step: {}".format(epoch, step))
  plt.savefig('image_at_epoch_{:04d}_step{:04d}.png'.format(epoch, step))
  plt.show()
```

You can now start the training loop. You are asked to select the number of epochs and to complete the subection on updating the weights. The general steps are:

- feed a training batch to the VAE model
- compute the reconstruction loss (hint: use the **mse_loss** defined above instead of `bce_loss` in the ungraded lab, then multiply by the flattened dimensions of the image (i.e. 64 x 64 x 3)
- add the KLD regularization loss to the total loss (you can access the `losses` property of the `vae` model)
- get the gradients
- use the optimizer to update the weights

When training your VAE, you might notice that there's not a lot of variation in the faces. But don't let that deter you! We'll test based on how well it does in reconstructing the original faces, and not how well it does in creating new faces.

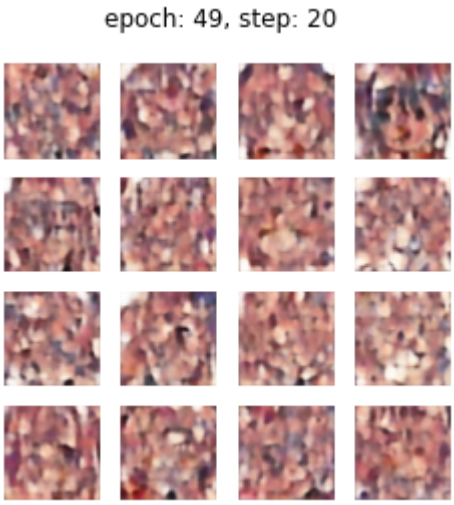
The training will also take a long time (more than 30 minutes) and that is to be expected. If you used the mean loss metric suggested above, train the model until that is down to around 320 before submitting.

```
BATCH_SIZE
```

```
     2000
```

```
# Training loop. Display generated images each epoch

### START CODE HERE ###
epochs = 50 #100 #32
### END CODE HERE ###
```

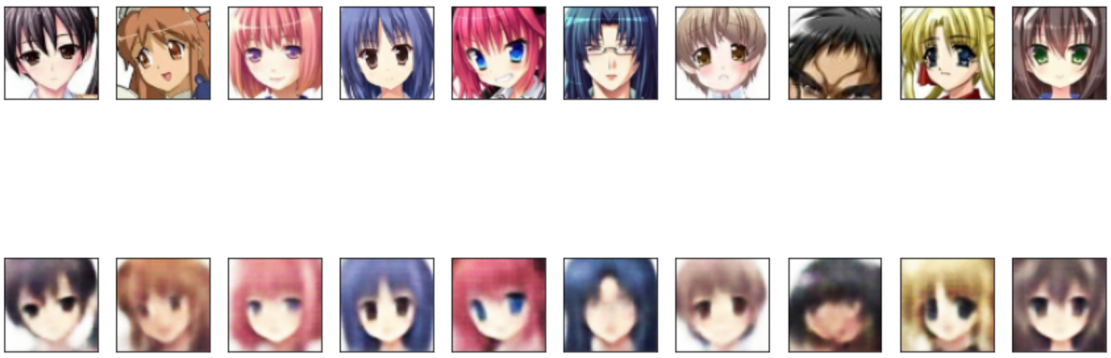
```
random_vector_for_generation = tf.random.normal(shape=[16, LATENT_DIM])
generate_and_save_images(decoder, 0, 0, random_vector_for_generation)

for epoch in range(epochs):
  print('Start of epoch %d' % (epoch,))

  # Iterate over the batches of the dataset.
  for step, x_batch_train in enumerate(training_dataset):
    with tf.GradientTape() as tape:
      ### START CODE HERE ###
      # feed a batch to the VAE model
      reconstructed = vae(x_batch_train)
      # Compute reconstruction loss
      flattened_inputs = tf.reshape(x_batch_train, shape=[-1])
      flattened_outputs = tf.reshape(reconstructed, shape=[-1])
      #loss = bce_loss(flattened_inputs, flattened_outputs) * 64*64*3
      loss = mse_loss(flattened_inputs, flattened_outputs) * 64*64*3

      # add KLD regularization loss
      loss += sum(vae.losses)

    grads = tape.gradient(loss, vae.trainable_weights)
    optimizer.apply_gradients(zip(grads, vae.trainable_weights))
    ### END CODE HERE

    loss_metric(loss)

    if step % 10 == 0:
      display.clear_output(wait=False)
      generate_and_save_images(decoder, epoch, step, random_vector_for_generation)
    print('Epoch: %s step: %s mean loss = %s' % (epoch, step, loss_metric.result().numpy()))
```



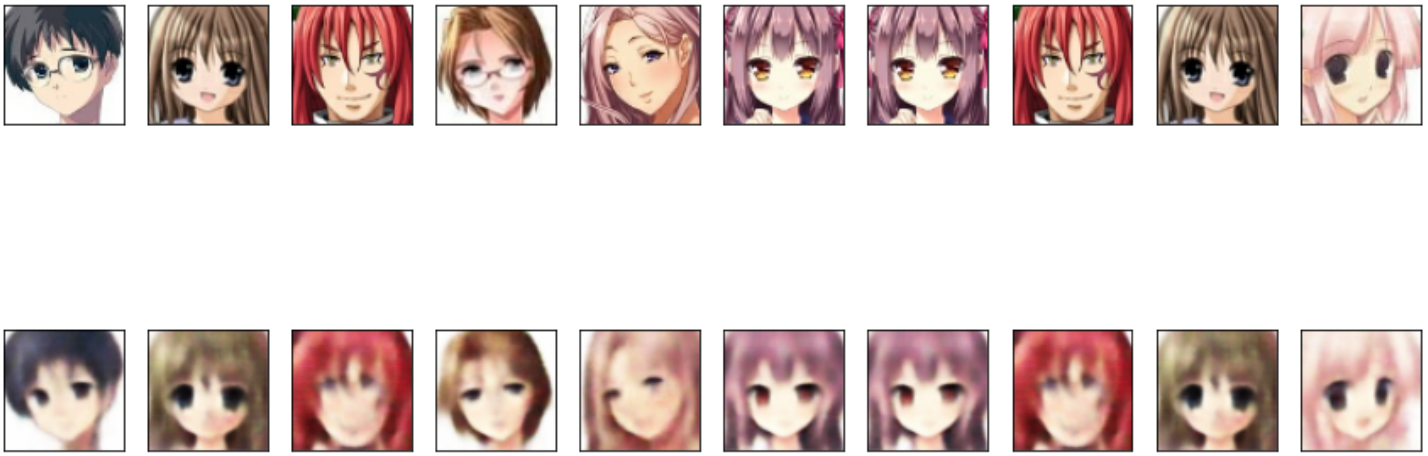

```
Epoch: 49 step: 20 mean loss = 230.95732
Epoch: 49 step: 21 mean loss = 230.9257
Epoch: 49 step: 22 mean loss = 230.8944
Epoch: 49 step: 23 mean loss = 230.86287
Epoch: 49 step: 24 mean loss = 230.82982
Epoch: 49 step: 25 mean loss = 230.79594
```

Plot Reconstructed Images

As mentioned, your model will be graded on how well it is able to reconstruct images (not generate new ones). You can get a glimpse of how it is doing with the code block below. It feeds in a batch from the test set and plots a row of input (top) and output (bottom) images. Don't worry if the outputs are a blurry. It will look something like below:

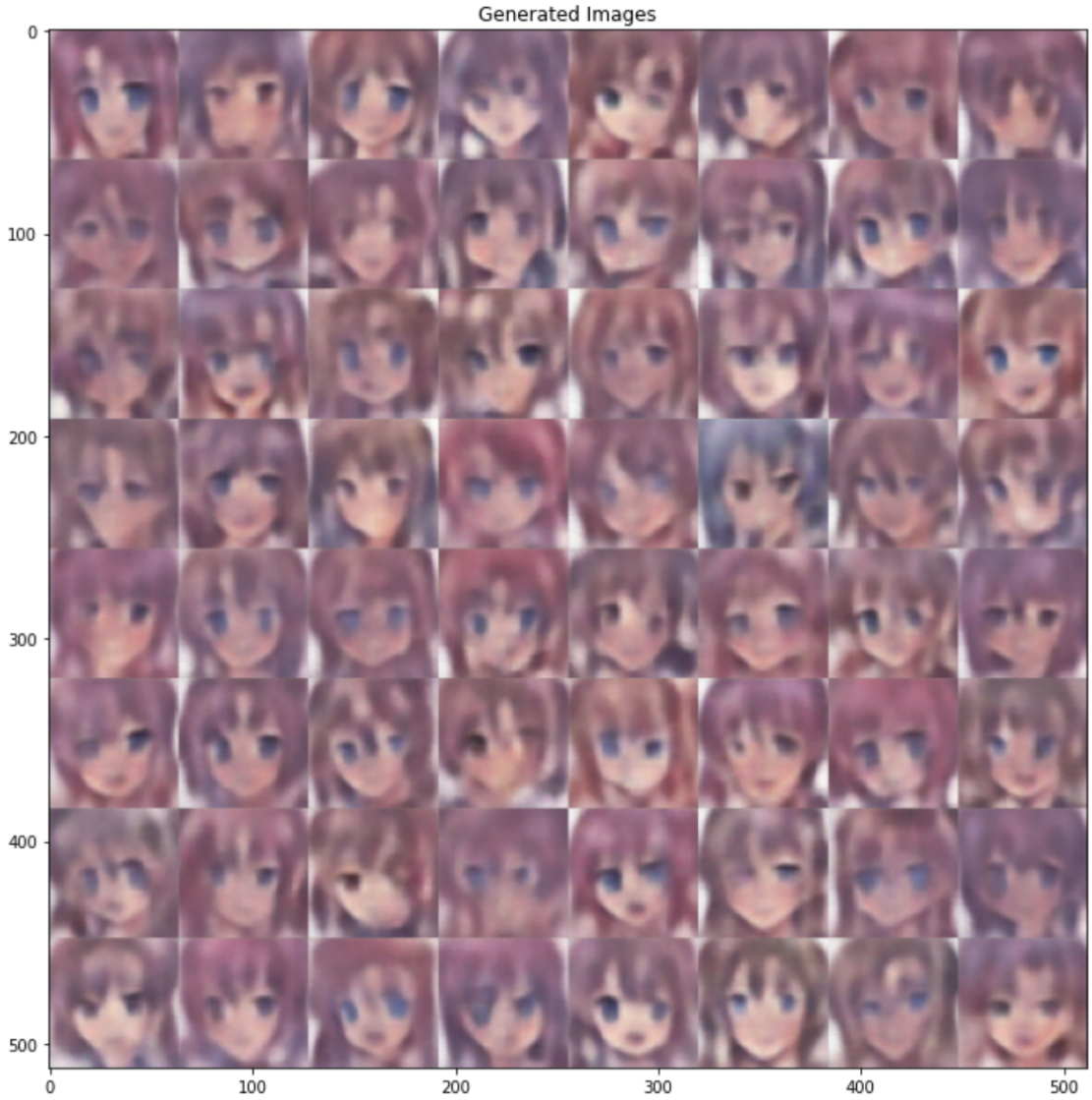


```
test_dataset = validation_dataset.take(1)
output_samples = []

for input_image in tfds.as_numpy(test_dataset):
      output_samples = input_image

idxs = np.random.choice(64, size=10)

vae_predicted = vae.predict(test_dataset)
display_results(output_samples[idxs], vae_predicted[idxs])
```

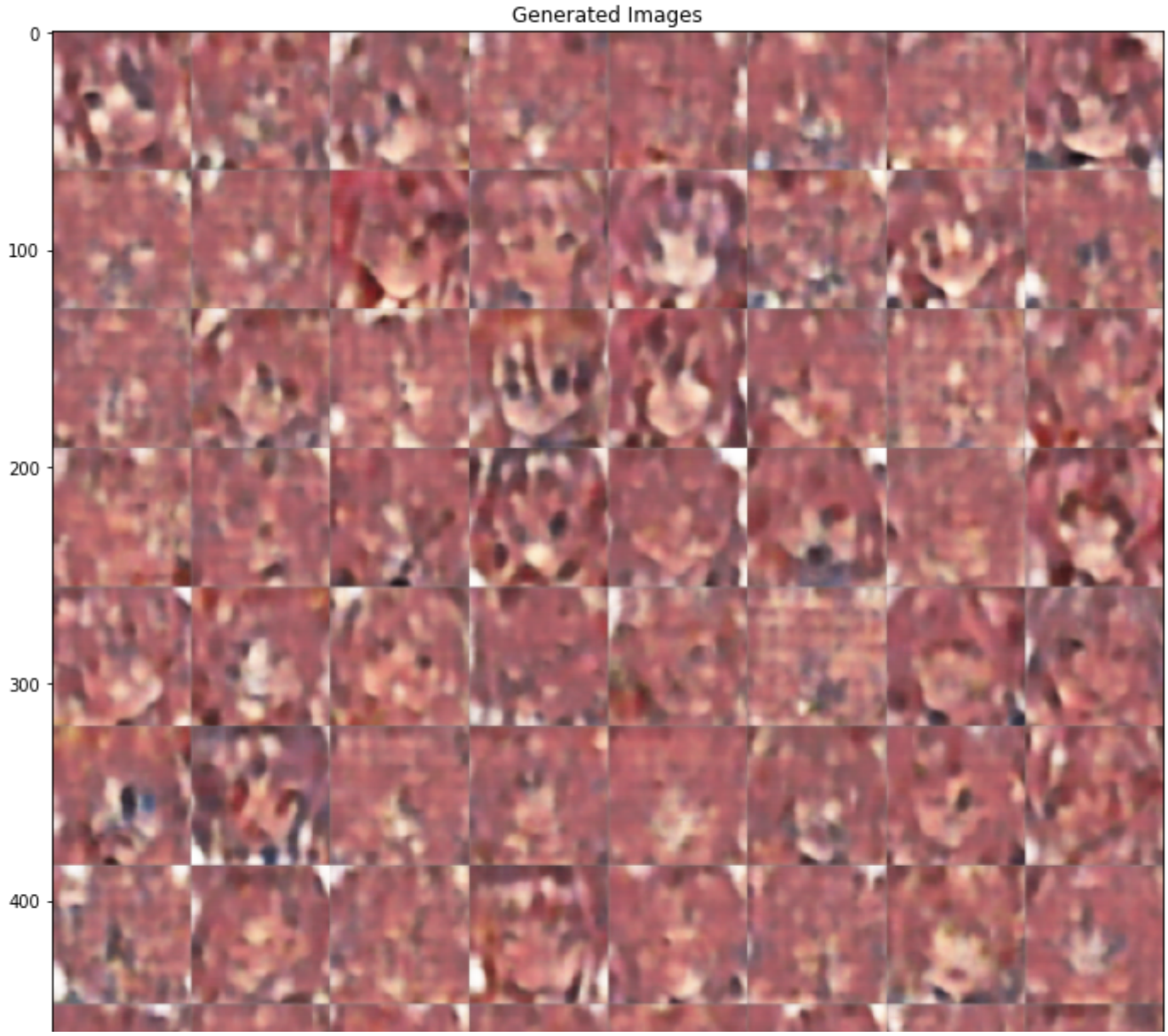


Plot Generated Images

Using the default parameters, it can take a long time to train your model well enough to generate good fake anime faces. In case you decide to experiment, we provided the code block below to display an 8x8 gallery of fake data generated from your model. Here is a sample gallery generated after 50 epochs.




```
def plot_images(rows, cols, images, title):
    '''Displays images in a grid.'''
    grid = np.zeros(shape=(rows*64, cols*64, 3))
    for row in range(rows):
        for col in range(cols):
            grid[row*64:(row+1)*64, col*64:(col+1)*64, :] = images[row*cols + col]

    plt.figure(figsize=(12,12))
    plt.imshow(grid)
    plt.title(title)
    plt.show()

# initialize random inputs
test_vector_for_generation = tf.random.normal(shape=[64, LATENT_DIM])

# get predictions from the decoder model
predictions= decoder.predict(test_vector_for_generation)

# plot the predictions
plot_images(8,8,predictions,'Generated Images')
```



Save the Model

Once your satisfied with the results, please save and download the model. Afterwards, please go back to the Coursera submission portal to upload your h5 file to the autograder.

```
vae.save("anime.h5")
```

✓ 1h 16m 21s completed at 10:54 AM