De La Salle University – Manila

Gokongwei College of Engineering

Department of Electronics and Computer Engineering

CpE Elective 3 Laboratory

LBYCPC4

Laboratory Report #3

Applications of Image Generation Models

by

Boncodin, Carl Patrick Q.

Chua, Kendrick Dayle J.

Concepcion, Edwin Jr. S.

Evaristo, Gier Bryant J.

Ong, Bryce Erwin D.

LBYCPC4 – EQ1

Introduction and Objectives

The discipline of machine learning made great progress in recent years with regard to image-generating models. These intricate deep-learning architectures showed significant potential in creating realistic, high-quality photos from images. This activity explored these models, concentrating on Super-Resolution Generative Adversarial Networks (SRGANs) and Conditional Generative Adversarial Networks (CGANs). These models facilitated the production of images conditioned on particular inputs and improved the resolution of images. By utilizing these advanced models, significant improvements were achieved in areas such as image restoration, enhancement, and generation, where clarity and precision were critical (GeeksforGeeks, 2024).

The activity clarified the basic ideas behind image-generating models, namely the design and workings of CGANs and SRGANs (Chakraborty, 2022). By adding additional information, like class labels, CGANs allowed for more controlled image generation by directing the development process toward desired results. SRGANs, on the other hand, produced high-resolution images from lower-resolution inputs to address the challenge of image super-resolution. The students created and trained these models, which helped them learn how to adjust parameters to achieve optimal results and visualize the images produced. Through this hands-on approach, they gained insights into the challenges of training GANs, such as balancing the generator and discriminator performances, and how to address issues like mode collapse and vanishing gradients (Chakraborty, 2022).

The main goal of this activity was for students to learn advanced image-generating methods by emphasizing the design and training of CGAN and SRGAN models. The activity also aimed to provide a thorough understanding of these complex image-creation techniques, covering both their theoretical foundation and real-world applications. Students experimented with model parameters and optimization strategies, deepening their understanding of the underlying mechanics of GANs and their practical uses in fields such as image enhancement, content creation, and medical imaging (GeeksforGeeks, 2024). Through this activity, the students were able to assess and visualize the outcomes of these models, appreciating both their potential and limitations in modern machine-learning tasks.

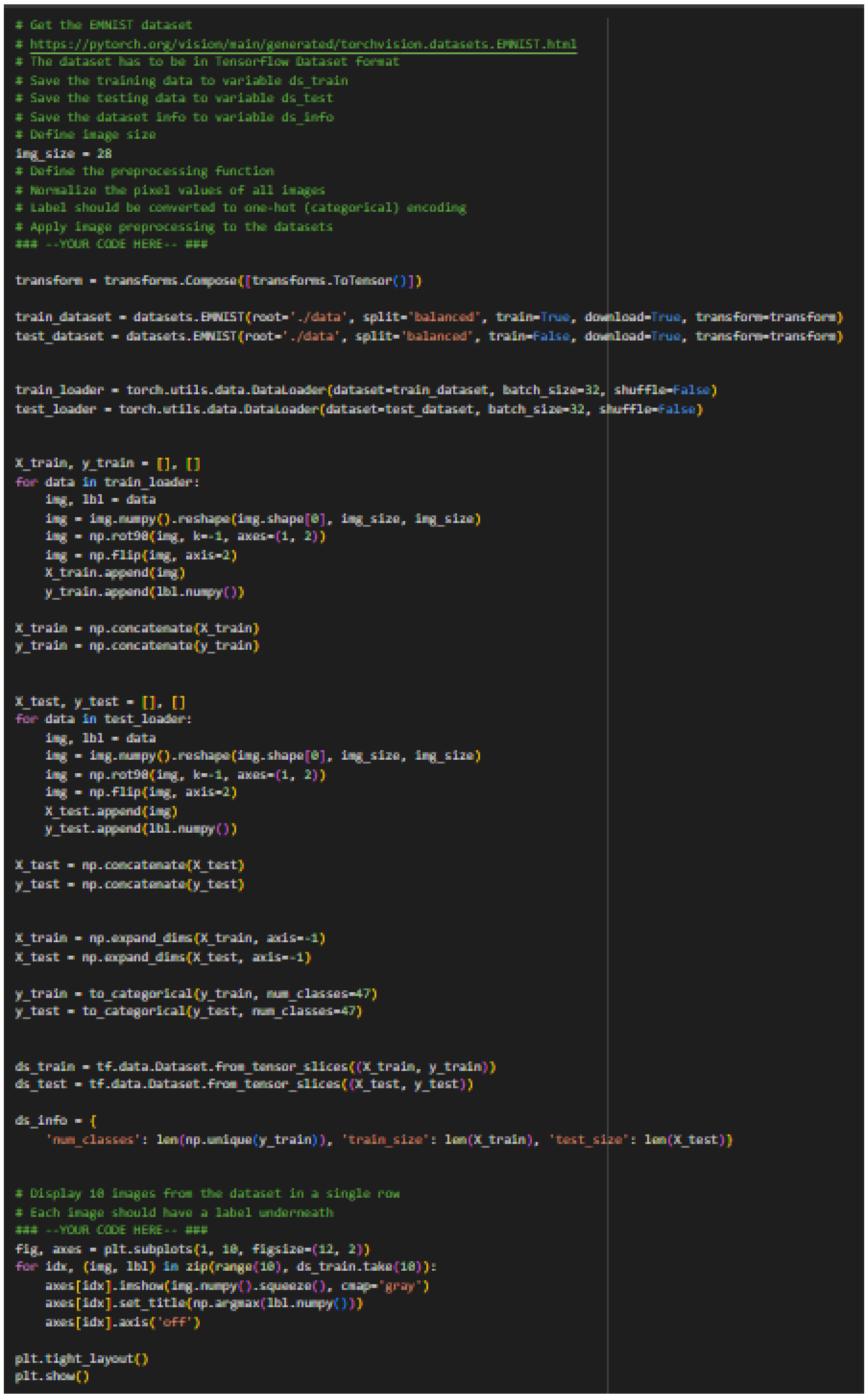
This activity aims to achieve the following objectives through code implementation in Python:

Objectives:

- Familiarize with the architecture of conditional GAN
- Familiarize the architecture of super-resolution GAN
- Build and train a conditional GAN image generative model using deep learning framework
- Build and train a super-resolution GAN image generative model using deep learning framework
- Visualize the output of image generation models
- Assess the performance of the implemented image generation models

Procedure/Methodology

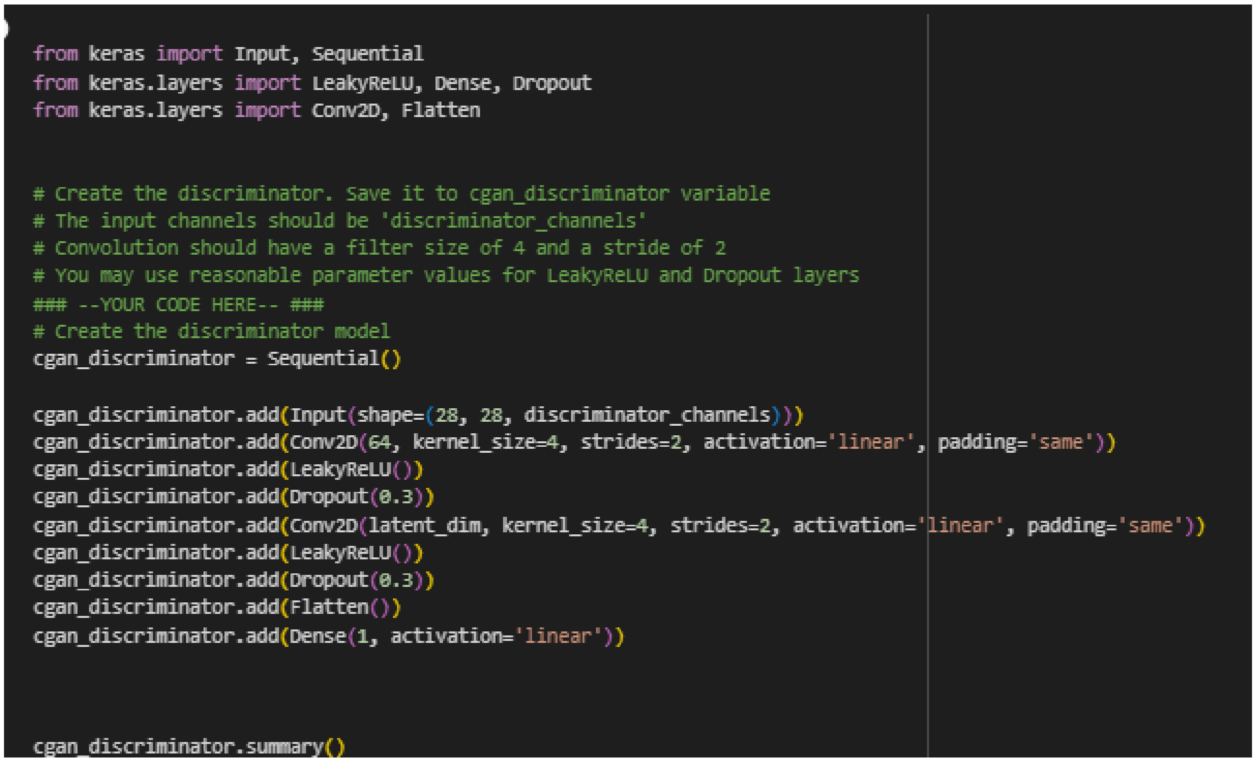
A. Controllable Image Generation using Conditional GAN



```python
# Get the EMNIST dataset
# https://pytorch.org/vision/main/generated/torchvision.datasets.EMNIST.html
# The dataset has to be in Tensorflow Dataset format
# Save the training data to variable ds_train
# Save the testing data to variable ds_test
# Save the dataset info to variable ds_info
# Define image size
img_size = 28
# Define the preprocessing function
# Normalize the pixel values of all images
# Label should be converted to one-hot (categorical) encoding
# Apply image preprocessing to the datasets
### --YOUR CODE HERE-- ###

transform = transforms.Compose([transforms.ToTensor()])

train_dataset = datasets.EMNIST(root='./data', split='balanced', train=True, download=True, transform=transform)
test_dataset = datasets.EMNIST(root='./data', split='balanced', train=False, download=True, transform=transform)


train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=32, shuffle=False)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=32, shuffle=False)


X_train, y_train = [], []
for data in train_loader:
    img, lbl = data
    img = img.numpy().reshape(img.shape[0], img_size, img_size)
    img = np.rot90(img, k=-1, axes=(1, 2))
    img = np.flip(img, axis=2)
    X_train.append(img)
    y_train.append(lbl.numpy())

X_train = np.concatenate(X_train)
y_train = np.concatenate(y_train)


X_test, y_test = [], []
for data in test_loader:
    img, lbl = data
    img = img.numpy().reshape(img.shape[0], img_size, img_size)
    img = np.rot90(img, k=-1, axes=(1, 2))
    img = np.flip(img, axis=2)
    X_test.append(img)
    y_test.append(lbl.numpy())

X_test = np.concatenate(X_test)
y_test = np.concatenate(y_test)


X_train = np.expand_dims(X_train, axis=-1)
X_test = np.expand_dims(X_test, axis=-1)

y_train = to_categorical(y_train, num_classes=47)
y_test = to_categorical(y_test, num_classes=47)


ds_train = tf.data.Dataset.from_tensor_slices((X_train, y_train))
ds_test = tf.data.Dataset.from_tensor_slices((X_test, y_test))

ds_info = {
    'num_classes': len(np.unique(y_train)), 'train_size': len(X_train), 'test_size': len(X_test)}


# Display 10 images from the dataset in a single row
# Each image should have a label underneath
### --YOUR CODE HERE-- ###
fig, axes = plt.subplots(1, 10, figsize=(12, 2))
for idx, (img, lbl) in zip(range(10), ds_train.take(10)):
    axes[idx].imshow(img.numpy().squeeze(), cmap='gray')
    axes[idx].set_title(np.argmax(lbl.numpy()))
    axes[idx].axis('off')

plt.tight_layout()
plt.show()
```

Figure 1. Code Snippet for Initializing and Displaying the EMNIST Dataset

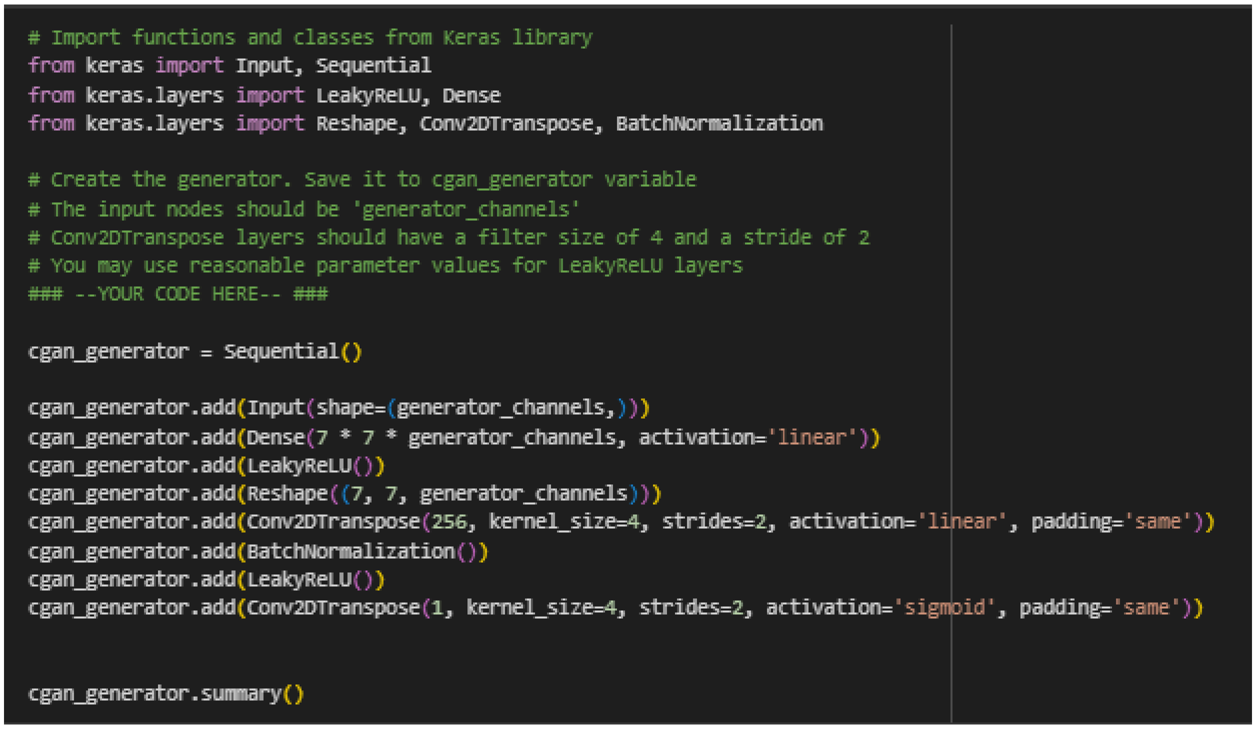
The code snippet in Figure 1 shows the code snippet for initializing and displaying the EMNIST Dataset. The dataset is first imported where the data is split into train and test. Once the train and test data sets are created, the datasets are processed, and all images are reshaped and rotated. All images and labels are appended to a list, which is converted into a numpy array. The labels are then converted into one hot encoding. The TensorFlow dataset for both training and testing is created using the code "tf.data.Dataset.from_tensor_slices()". A dictionary is then created, which contains information about the dataset. Lastly, 10 images from the dataset are then displayed in a single row.



```
from keras import Input, Sequential
from keras.layers import LeakyReLU, Dense, Dropout
from keras.layers import Conv2D, Flatten


# Create the discriminator. Save it to cgan_discriminator variable
# The input channels should be 'discriminator_channels'
# Convolution should have a filter size of 4 and a stride of 2
# You may use reasonable parameter values for LeakyReLU and Dropout layers
### --YOUR CODE HERE-- ###
# Create the discriminator model
cgan_discriminator = Sequential()

cgan_discriminator.add(Input(shape=(28, 28, discriminator_channels)))
cgan_discriminator.add(Conv2D(64, kernel_size=4, strides=2, activation='linear', padding='same'))
cgan_discriminator.add(LeakyReLU())
cgan_discriminator.add(Dropout(0.3))
cgan_discriminator.add(Conv2D(latent_dim, kernel_size=4, strides=2, activation='linear', padding='same'))
cgan_discriminator.add(LeakyReLU())
cgan_discriminator.add(Dropout(0.3))
cgan_discriminator.add(Flatten())
cgan_discriminator.add(Dense(1, activation='linear'))



cgan_discriminator.summary()
```

Figure 2. Code Snippet for the Creation of Discriminator

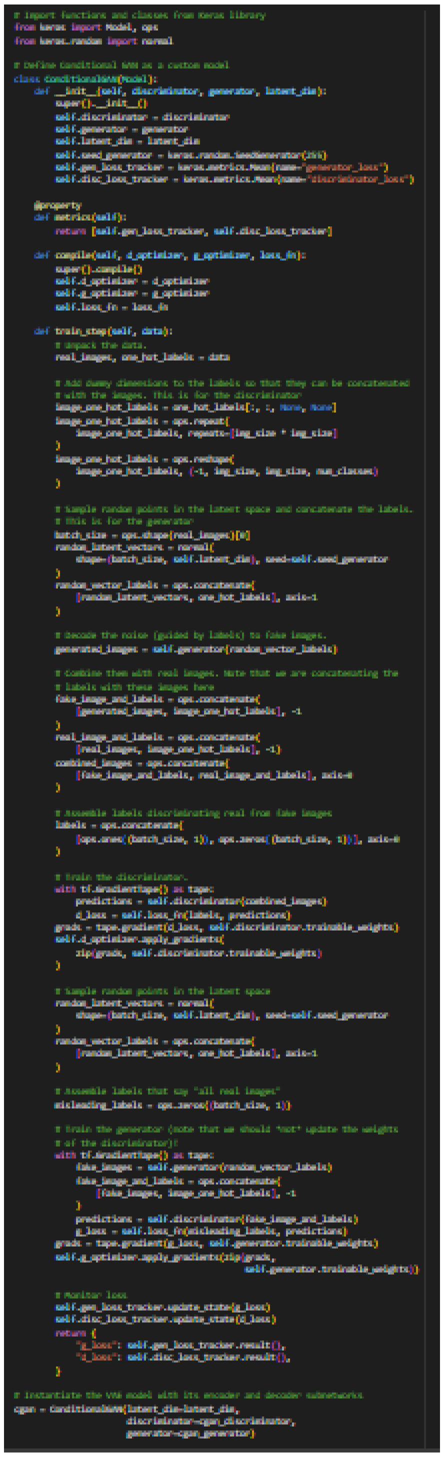
The code snippet in Figure 2 shows the creation of the discriminator. The code starts by importing the important libraries that create the discriminator. Once the necessary libraries are imported, the discriminator is then created using the specifications given in the activity. The model summary is then shown to verify whether the architecture of the discriminator is the same as the specifications provided.



```python
# Import functions and classes from Keras library
from keras import Input, Sequential
from keras.layers import LeakyReLU, Dense
from keras.layers import Reshape, Conv2DTranspose, BatchNormalization

# Create the generator. Save it to cgan_generator variable
# The input nodes should be 'generator_channels'
# Conv2DTranspose layers should have a filter size of 4 and a stride of 2
# You may use reasonable parameter values for LeakyReLU layers
### --YOUR CODE HERE-- ###

cgan_generator = Sequential()

cgan_generator.add(Input(shape=(generator_channels,)))
cgan_generator.add(Dense(7 * 7 * generator_channels, activation='linear'))
cgan_generator.add(LeakyReLU())
cgan_generator.add(Reshape((7, 7, generator_channels)))
cgan_generator.add(Conv2DTranspose(256, kernel_size=4, strides=2, activation='linear', padding='same'))
cgan_generator.add(BatchNormalization())
cgan_generator.add(LeakyReLU())
cgan_generator.add(Conv2DTranspose(1, kernel_size=4, strides=2, activation='sigmoid', padding='same'))


cgan_generator.summary()
```

Figure 3. Code Snippet for the Creation of the Generator

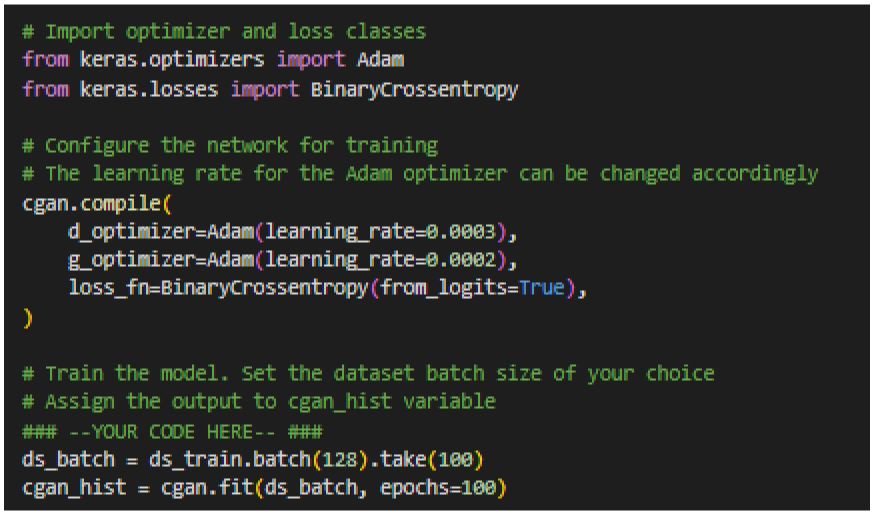
The code snippet in Figure 3 shows the creation of the generator. The code is similar as the previous code where it starts by importing the important libraries that create the generator. Once the necessary libraries are imported, the generator is then created using the specifications given in the activity. The model summary is then shown to verify whether the architecture of the generator is the same as the specifications provided.



```python
# Import functions and classes from Keras library
from keras import Model, ops
from keras.random import normal

# Define Conditional GAN as a custom model
class ConditionalGAN(Model):
    def __init__(self, discriminator, generator, latent_dim):
        super().__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.seed_generator = keras.random.SeedGenerator(1337)
        self.gen_loss_tracker = keras.metrics.Mean(name="generator_loss")
        self.disc_loss_tracker = keras.metrics.Mean(name="discriminator_loss")

    @property
    def metrics(self):
        return [self.gen_loss_tracker, self.disc_loss_tracker]

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super().compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.loss_fn = loss_fn

    def train_step(self, data):
        # Unpack the data.
        real_images, one_hot_labels = data

        # Add dummy dimensions to the labels so that they can be concatenated
        # with the images. This is for the discriminator
        image_one_hot_labels = one_hot_labels[:, :, None, None]
        image_one_hot_labels = ops.repeat(
            image_one_hot_labels, repeats=[img_size * img_size]
        )
        image_one_hot_labels = ops.reshape(
            image_one_hot_labels, (-1, img_size, img_size, num_classes)
        )

        # Sample random points in the latent space and concatenate the labels.
        # This is for the generator
        batch_size = ops.shape(real_images)[0]
        random_latent_vectors = normal(
            shape=(batch_size, self.latent_dim), seed=self.seed_generator
        )
        random_vector_labels = ops.concatenate(
            [random_latent_vectors, one_hot_labels], axis=1
        )

        # Decode the noise (guided by labels) to fake images.
        generated_images = self.generator(random_vector_labels)

        # Combine them with real images. Note that we are concatenating the
        # labels with these images here
        fake_image_and_labels = ops.concatenate(
            [generated_images, image_one_hot_labels], -1
        )
        real_image_and_labels = ops.concatenate(
            [real_images, image_one_hot_labels], -1)
        combined_images = ops.concatenate(
            [fake_image_and_labels, real_image_and_labels], axis=0
        )

        # Assemble labels discriminating real from fake images
        labels = ops.concatenate(
            [ops.ones((batch_size, 1)), ops.zeros((batch_size, 1))], axis=0
        )

        # Train the discriminator.
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply_gradients(
            zip(grads, self.discriminator.trainable_weights)
        )

        # Sample random points in the latent space
        random_latent_vectors = normal(
            shape=(batch_size, self.latent_dim), seed=self.seed_generator
        )
        random_vector_labels = ops.concatenate(
            [random_latent_vectors, one_hot_labels], axis=1
        )

        # Assemble labels that say "all real images"
        misleading_labels = ops.zeros((batch_size, 1))

        # Train the generator (note that we should *not* update the weights
        # of the discriminator)!
        with tf.GradientTape() as tape:
            fake_images = self.generator(random_vector_labels)
            fake_image_and_labels = ops.concatenate(
                [fake_images, image_one_hot_labels], -1
            )
            predictions = self.discriminator(fake_image_and_labels)
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(zip(grads,
                                             self.generator.trainable_weights))

        # Monitor loss
        self.gen_loss_tracker.update_state(g_loss)
        self.disc_loss_tracker.update_state(d_loss)
        return {
            "g_loss": self.gen_loss_tracker.result(),
            "d_loss": self.disc_loss_tracker.result(),
        }

# Instantiate the VAE model with its encoder and decoder subnetworks
cgan = ConditionalGAN(latent_dim=latent_dim,
                      discriminator=cgan_discriminator,
                      generator=cgan_generator)
```

Figure 4. Code Snippet for the Defining the Model

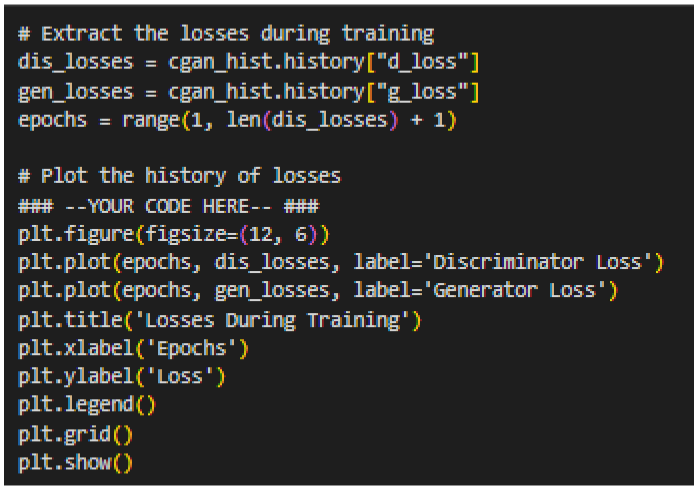
The code snippet in Figure 4 shows the definition of the class and the instantiation of its class. The provided code is provided where the code is executed, which defines the custom model and instantiates the Variational Autoencoder.



```python
# Import optimizer and loss classes
from keras.optimizers import Adam
from keras.losses import BinaryCrossentropy

# Configure the network for training
# The learning rate for the Adam optimizer can be changed accordingly
cgan.compile(
    d_optimizer=Adam(learning_rate=0.0003),
    g_optimizer=Adam(learning_rate=0.0002),
    loss_fn=BinaryCrossentropy(from_logits=True),
)

# Train the model. Set the dataset batch size of your choice
# Assign the output to cgan_hist variable
### --YOUR CODE HERE-- ###
ds_batch = ds_train.batch(128).take(100)
cgan_hist = cgan.fit(ds_batch, epochs=100)
```

Figure 5. Code Snippet for Training the Model

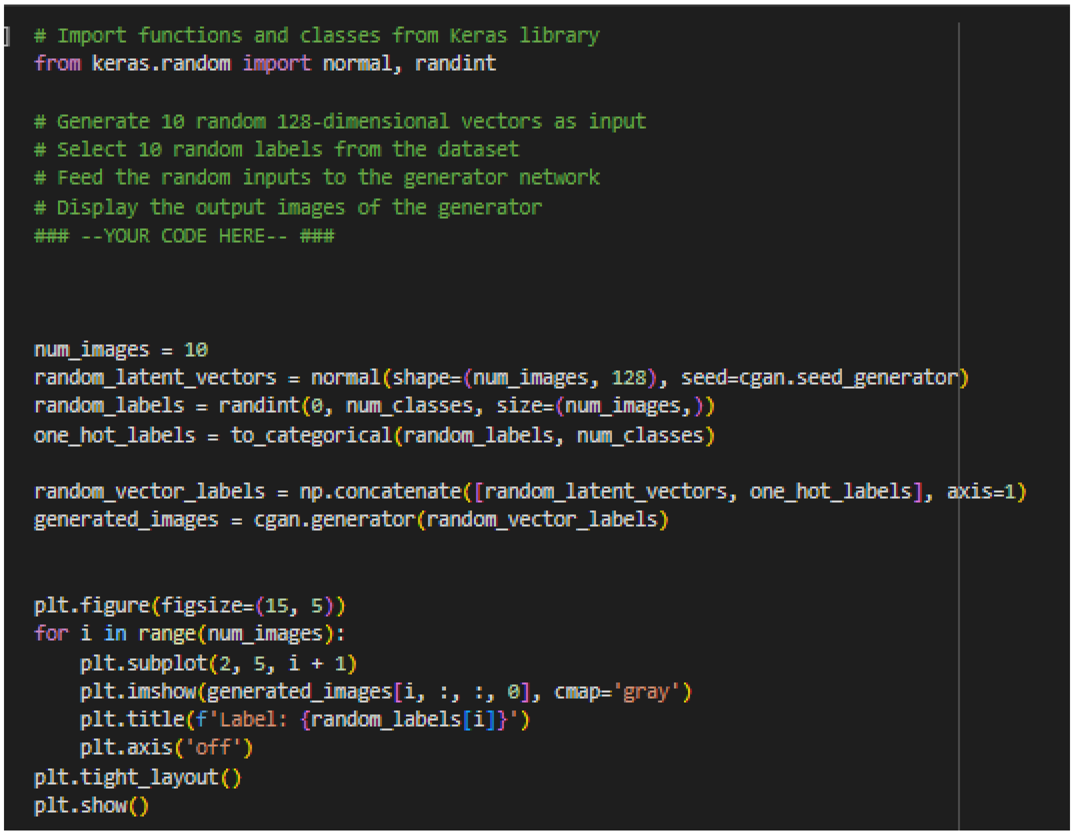
The code snippet in Figure 5 shows the configuration of the network for training. Parameters such as the optimizer and learning rate for both the discriminator and generator are configured, as well as the lost function. Once the network is configured, the model is then trained for 100 epochs with a batch size of 100.



```python
# Extract the losses during training
dis_losses = cgan_hist.history["d_loss"]
gen_losses = cgan_hist.history["g_loss"]
epochs = range(1, len(dis_losses) + 1)

# Plot the history of losses
### --YOUR CODE HERE-- ###
plt.figure(figsize=(12, 6))
plt.plot(epochs, dis_losses, label='Discriminator Loss')
plt.plot(epochs, gen_losses, label='Generator Loss')
plt.title('Losses During Training')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()
plt.show()
```

Figure 6. Code Snippet for Plotting the Generator and Discriminator Loss

The code snippet in Figure 6 shows the code for plotting both the Generator and Discriminator Loss. The Generator and Discriminator Loss are stored to gen_lossess and dis_lossess, respectively. The number of epochs used to train the model is also stored in the variable epochs. Once all necessary information is obtained, the Generator and Discriminator Loss are plotted.

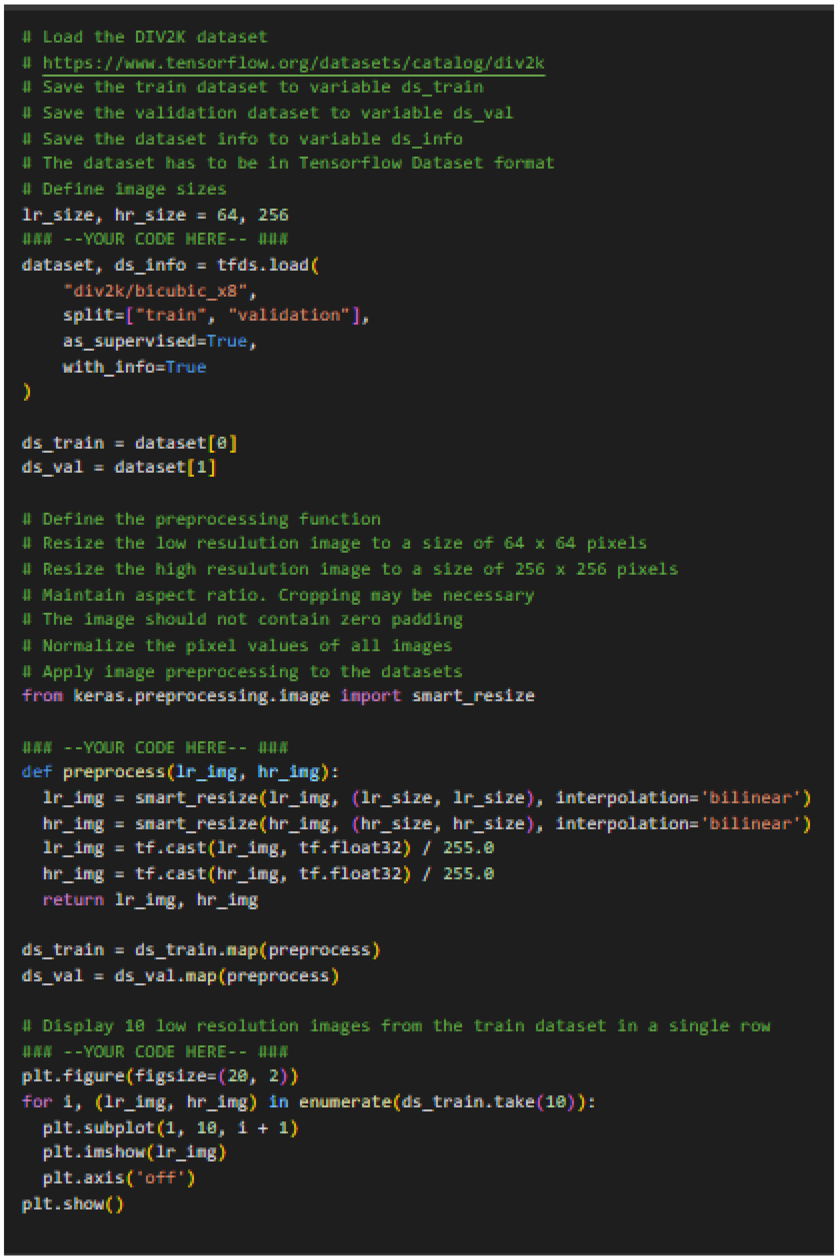


```
# Import functions and classes from Keras library
from keras.random import normal, randint

# Generate 10 random 128-dimensional vectors as input
# Select 10 random labels from the dataset
# Feed the random inputs to the generator network
# Display the output images of the generator
### --YOUR CODE HERE-- ###



num_images = 10
random_latent_vectors = normal(shape=(num_images, 128), seed=cgan.seed_generator)
random_labels = randint(0, num_classes, size=(num_images,))
one_hot_labels = to_categorical(random_labels, num_classes)

random_vector_labels = np.concatenate([random_latent_vectors, one_hot_labels], axis=1)
generated_images = cgan.generator(random_vector_labels)


plt.figure(figsize=(15, 5))
for i in range(num_images):
    plt.subplot(2, 5, i + 1)
    plt.imshow(generated_images[i, :, :, 0], cmap='gray')
    plt.title(f'Label: {random_labels[i]}')
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Figure 7. Code Snippet for Plotting the Generation of 10 Images

The code snippet in Figure 7 shows the code for generating 10 random images which are displayed. The code first generates 10 random 128-dimensional vectors and 10 random labels from the dataset. The labels are then converted to one_hot labels. The random vectors and the labels are concatenated into a numpy array, which is fed to the generator. Once fed, the generated images of the generator are displayed.

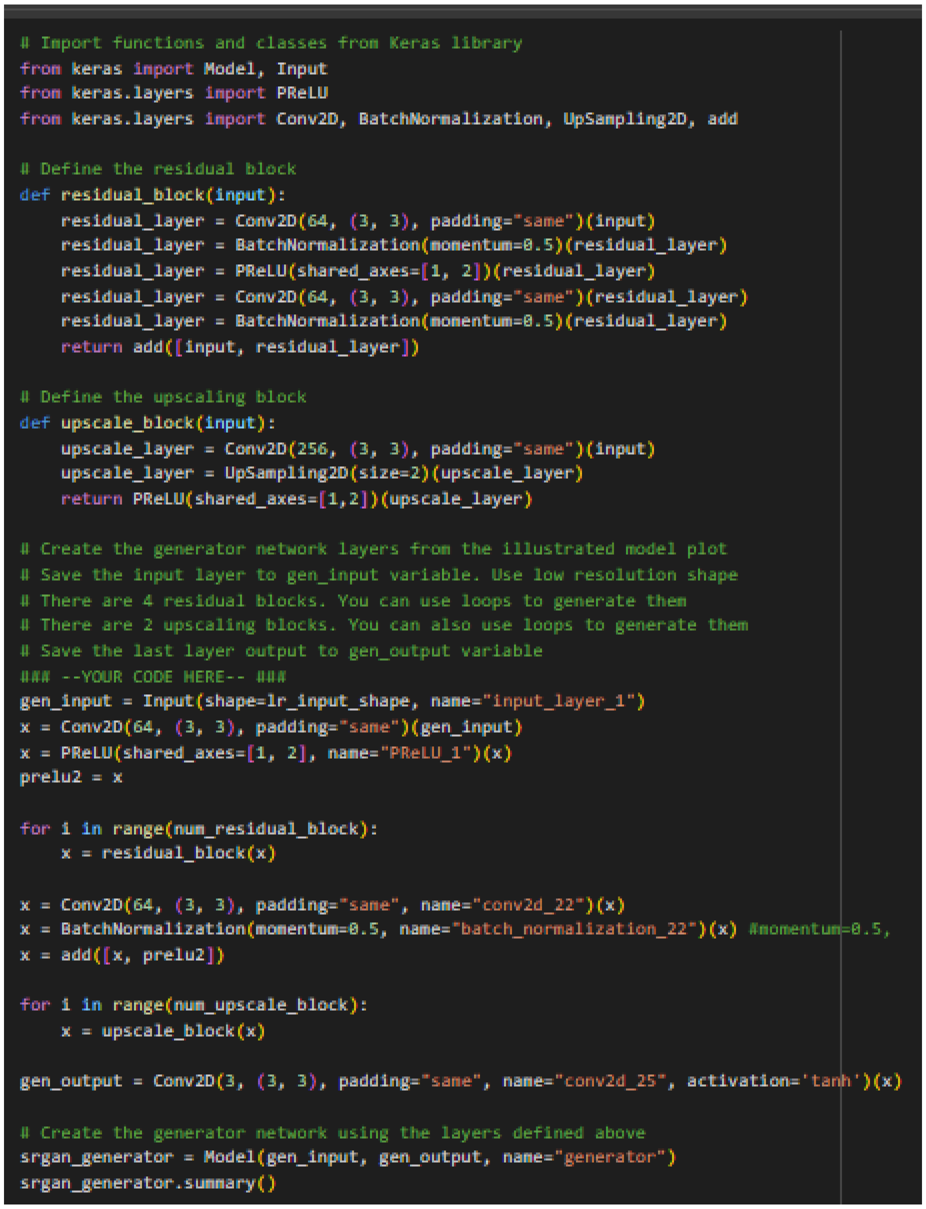
B. Image Super-Resolution using SRGAN



```
# Load the DIV2K dataset
# https://www.tensorflow.org/datasets/catalog/div2k
# Save the train dataset to variable ds_train
# Save the validation dataset to variable ds_val
# Save the dataset info to variable ds_info
# The dataset has to be in Tensorflow Dataset format
# Define image sizes
lr_size, hr_size = 64, 256
### --YOUR CODE HERE-- ###
dataset, ds_info = tfds.load(
    "div2k/bicubic_x8",
    split=["train", "validation"],
    as_supervised=True,
    with_info=True
)

ds_train = dataset[0]
ds_val = dataset[1]

# Define the preprocessing function
# Resize the low resulution image to a size of 64 x 64 pixels
# Resize the high resulution image to a size of 256 x 256 pixels
# Maintain aspect ratio. Cropping may be necessary
# The image should not contain zero padding
# Normalize the pixel values of all images
# Apply image preprocessing to the datasets
from keras.preprocessing.image import smart_resize

### --YOUR CODE HERE-- ###
def preprocess(lr_img, hr_img):
  lr_img = smart_resize(lr_img, (lr_size, lr_size), interpolation='bilinear')
  hr_img = smart_resize(hr_img, (hr_size, hr_size), interpolation='bilinear')
  lr_img = tf.cast(lr_img, tf.float32) / 255.0
  hr_img = tf.cast(hr_img, tf.float32) / 255.0
  return lr_img, hr_img

ds_train = ds_train.map(preprocess)
ds_val = ds_val.map(preprocess)

# Display 10 low resolution images from the train dataset in a single row
### --YOUR CODE HERE-- ###
plt.figure(figsize=(20, 2))
for i, (lr_img, hr_img) in enumerate(ds_train.take(10)):
  plt.subplot(1, 10, i + 1)
  plt.imshow(lr_img)
  plt.axis('off')
plt.show()
```

Figure 8. Code Snippet for Initializing and Displaying the DIV2K Dataset

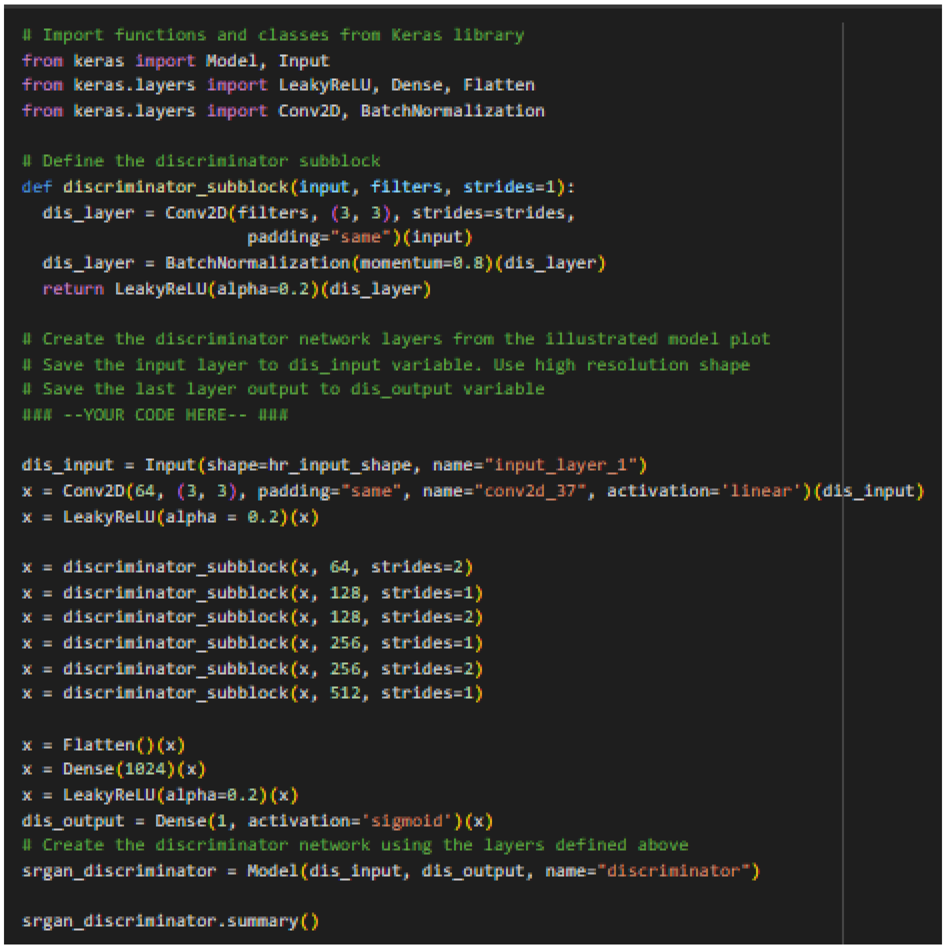
The code snippet in Figure 8 shows the code snippet for initializing and displaying the DIV2K Dataset. The dataset is first imported is split into train and validation. Once the training and validation data sets are created, the datasets are processed, and all images are resized to match the resolution. The high-resolution images are resized to a shape of (256,256) while the low-resolution images are resized to (64,64). The aspect ratio of the images is preserved, and the pixel values of all images are normalized. Once all images are processed, 10 low-resolution images are displayed in a single row.



```python
# Import functions and classes from Keras library
from keras import Model, Input
from keras.layers import PReLU
from keras.layers import Conv2D, BatchNormalization, UpSampling2D, add

# Define the residual block
def residual_block(input):
    residual_layer = Conv2D(64, (3, 3), padding="same")(input)
    residual_layer = BatchNormalization(momentum=0.5)(residual_layer)
    residual_layer = PReLU(shared_axes=[1, 2])(residual_layer)
    residual_layer = Conv2D(64, (3, 3), padding="same")(residual_layer)
    residual_layer = BatchNormalization(momentum=0.5)(residual_layer)
    return add([input, residual_layer])

# Define the upscaling block
def upscale_block(input):
    upscale_layer = Conv2D(256, (3, 3), padding="same")(input)
    upscale_layer = UpSampling2D(size=2)(upscale_layer)
    return PReLU(shared_axes=[1,2])(upscale_layer)

# Create the generator network layers from the illustrated model plot
# Save the input layer to gen_input variable. Use low resolution shape
# There are 4 residual blocks. You can use loops to generate them
# There are 2 upscaling blocks. You can also use loops to generate them
# Save the last layer output to gen_output variable
### --YOUR CODE HERE-- ###
gen_input = Input(shape=lr_input_shape, name="input_layer_1")
x = Conv2D(64, (3, 3), padding="same")(gen_input)
x = PReLU(shared_axes=[1, 2], name="PReLU_1")(x)
prelu2 = x

for i in range(num_residual_block):
    x = residual_block(x)

x = Conv2D(64, (3, 3), padding="same", name="conv2d_22")(x)
x = BatchNormalization(momentum=0.5, name="batch_normalization_22")(x) #momentum=0.5,
x = add([x, prelu2])

for i in range(num_upscale_block):
    x = upscale_block(x)

gen_output = Conv2D(3, (3, 3), padding="same", name="conv2d_25", activation='tanh')(x)

# Create the generator network using the layers defined above
srgan_generator = Model(gen_input, gen_output, name="generator")
srgan_generator.summary()
```

Figure 8. Code Snippet for the Creation of the Generator

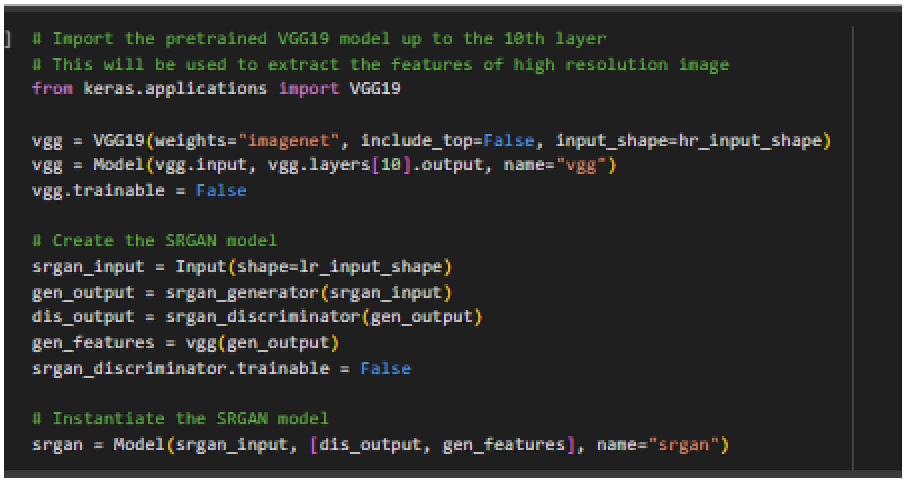
The code snippet in Figure 8 shows the creation of the generator. The code starts by importing the important libraries that create the generator. Two functions are created, which are used to create a block of network layers for the generator. The input layer of the generator is first created followed by the Conv2D and the PreLu layer. Once the first three layers of the generator are created, a for loop is created, which uses the provided functions to create the residual blocks and upscale blocks in the generator. Once the generator is created, the generator summary is then shown to verify whether the architecture of the generator is the same as the specifications provided.



```
# Import functions and classes from Keras library
from keras import Model, Input
from keras.layers import LeakyReLU, Dense, Flatten
from keras.layers import Conv2D, BatchNormalization

# Define the discriminator subblock
def discriminator_subblock(input, filters, strides=1):
  dis_layer = Conv2D(filters, (3, 3), strides=strides,
                     padding="same")(input)
  dis_layer = BatchNormalization(momentum=0.8)(dis_layer)
  return LeakyReLU(alpha=0.2)(dis_layer)

# Create the discriminator network layers from the illustrated model plot
# Save the input layer to dis_input variable. Use high resolution shape
# Save the last layer output to dis_output variable
### --YOUR CODE HERE-- ###

dis_input = Input(shape=hr_input_shape, name="input_layer_1")
x = Conv2D(64, (3, 3), padding="same", name="conv2d_37", activation='linear')(dis_input)
x = LeakyReLU(alpha = 0.2)(x)

x = discriminator_subblock(x, 64, strides=2)
x = discriminator_subblock(x, 128, strides=1)
x = discriminator_subblock(x, 128, strides=2)
x = discriminator_subblock(x, 256, strides=1)
x = discriminator_subblock(x, 256, strides=2)
x = discriminator_subblock(x, 512, strides=1)

x = Flatten()(x)
x = Dense(1024)(x)
x = LeakyReLU(alpha=0.2)(x)
dis_output = Dense(1, activation='sigmoid')(x)
# Create the discriminator network using the layers defined above
srgan_discriminator = Model(dis_input, dis_output, name="discriminator")

srgan_discriminator.summary()
```

Figure 9. Code Snippet for the Creation of the Discriminator

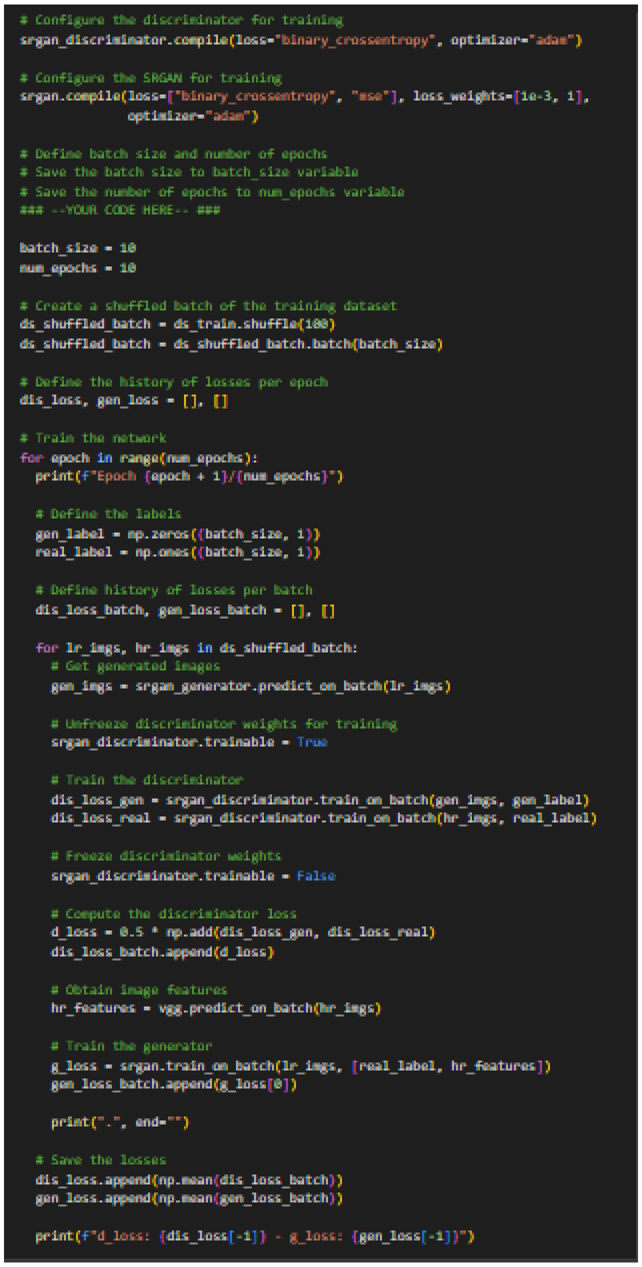
The code snippet in Figure 9 shows the creation of the discriminator. The code starts by importing the important libraries that create the discriminator. a function is provided that is used to create a block of network layers for the discriminator. The input layer of the discriminator is first created followed by the Conv2D and the LeakyReLu layer. Once the first three layers of the discriminator are created, the function discriminator_subblock is called 6 times with different values for filters and strides. The final layers of the discriminator, which are flatten, dense, LeakyReLU, and output layer are created. Once all the network layers of the discriminator are defined, T=the discriminator is then created. The discriminator summary is then shown to verify whether the architecture of the discriminator is the same as the specifications provided.



```
# Import the pretrained VGG19 model up to the 10th layer
# This will be used to extract the features of high resolution image
from keras.applications import VGG19

vgg = VGG19(weights="imagenet", include_top=False, input_shape=hr_input_shape)
vgg = Model(vgg.input, vgg.layers[10].output, name="vgg")
vgg.trainable = False

# Create the SRGAN model
srgan_input = Input(shape=lr_input_shape)
gen_output = srgan_generator(srgan_input)
dis_output = srgan_discriminator(gen_output)
gen_features = vgg(gen_output)
srgan_discriminator.trainable = False

# Instantiate the SRGAN model
srgan = Model(srgan_input, [dis_output, gen_features], name="srgan")
```

Figure 10. Code Snippet for the Creation of the SRGAN Model

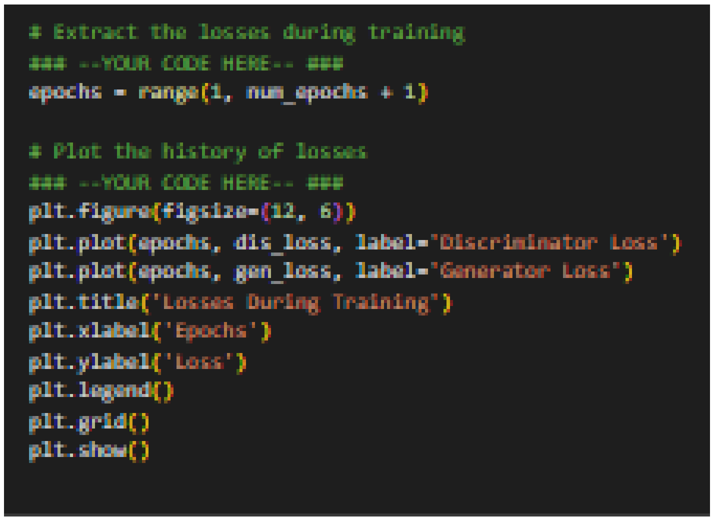
The code snippet shows the creation of the SRGAN model which will be used to extract features of high-resolution images. The necessary library for the VGG19 is first imported in order to instantiate the VGG19 model. Once the required libraries are imported, a VGG19 model is instantiated. Once the instantiation of the VGG19 model is created, the network layers of the SRGAN are defined where the SRGAN would be instantiated.



```python
# Configure the discriminator for training
srgan_discriminator.compile(loss="binary_crossentropy", optimizer="adam")

# Configure the SRGAN for training
srgan.compile(loss=["binary_crossentropy", "mse"], loss_weights=[1e-3, 1],
              optimizer="adam")

# Define batch size and number of epochs
# Save the batch size to batch_size variable
# Save the number of epochs to num_epochs variable
### --YOUR CODE HERE-- ###

batch_size = 10
num_epochs = 10

# Create a shuffled batch of the training dataset
ds_shuffled_batch = ds_train.shuffle(100)
ds_shuffled_batch = ds_shuffled_batch.batch(batch_size)

# Define the history of losses per epoch
dis_loss, gen_loss = [], []

# Train the network
for epoch in range(num_epochs):
  print(f"Epoch {epoch + 1}/{num_epochs}")

  # Define the labels
  gen_label = np.zeros((batch_size, 1))
  real_label = np.ones((batch_size, 1))

  # Define history of losses per batch
  dis_loss_batch, gen_loss_batch = [], []

  for lr_imgs, hr_imgs in ds_shuffled_batch:
    # Get generated images
    gen_imgs = srgan_generator.predict_on_batch(lr_imgs)

    # Unfreeze discriminator weights for training
    srgan_discriminator.trainable = True

    # Train the discriminator
    dis_loss_gen = srgan_discriminator.train_on_batch(gen_imgs, gen_label)
    dis_loss_real = srgan_discriminator.train_on_batch(hr_imgs, real_label)

    # Freeze discriminator weights
    srgan_discriminator.trainable = False

    # Compute the discriminator loss
    d_loss = 0.5 * np.add(dis_loss_gen, dis_loss_real)
    dis_loss_batch.append(d_loss)

    # Obtain image features
    hr_features = vgg.predict_on_batch(hr_imgs)

    # Train the generator
    g_loss = srgan.train_on_batch(lr_imgs, [real_label, hr_features])
    gen_loss_batch.append(g_loss[0])

    print(".", end="")

  # Save the losses
  dis_loss.append(np.mean(dis_loss_batch))
  gen_loss.append(np.mean(gen_loss_batch))

  print(f"d_loss: {dis_loss[-1]} - g_loss: {gen_loss[-1]}")
```

Figure 11. Code Snippet for Training Discriminator and SRGAN

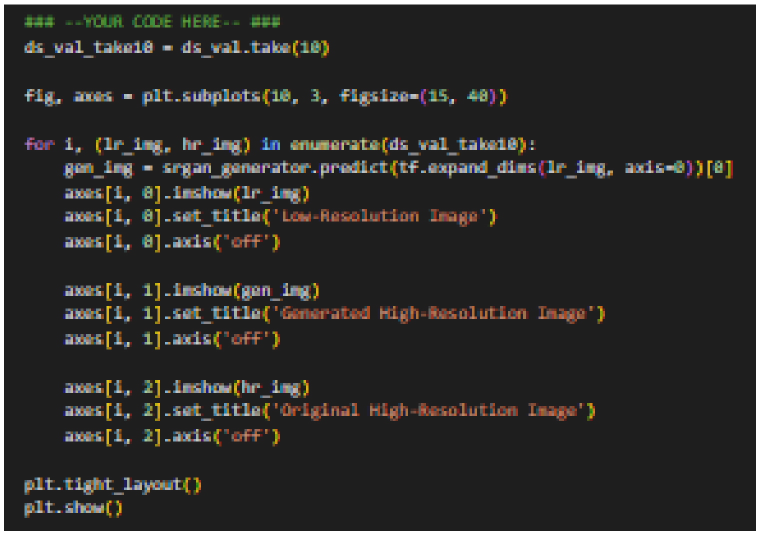
The code snippet in Figure 11 shows the training of the discriminator and SRGAN. The first step is to configure both the discriminator and the SRGAN for training. Once configured, the batch size and number of epochs are defined, and a shuffle batch of the training dataset is created. The training of both the discriminator and SRGan is executed where the discriminator would be unfrozen and trained with both real and generated data. Once trained, the discriminator's weights are frozen, and the discriminator loss is computer. The generator would then be trained with the extracted image features. The losses are then saved and printed.



```python
# Extract the losses during training
### --YOUR CODE HERE-- ###
epochs = range(1, num_epochs + 1)

# Plot the history of losses
### --YOUR CODE HERE-- ###
plt.figure(figsize=(12, 6))
plt.plot(epochs, dis_loss, label='Discriminator Loss')
plt.plot(epochs, gen_loss, label='Generator Loss')
plt.title('Losses During Training')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()
plt.show()
```

Figure 12. Code Snippet for the Plotting of Discriminator and Generator Losses

The code snippet in Figure 12 shows the code for plotting both the discriminator loss and the generator loss. Using the same discriminator and generator losses from the previous code, the losses are then plotted over the number of epochs.



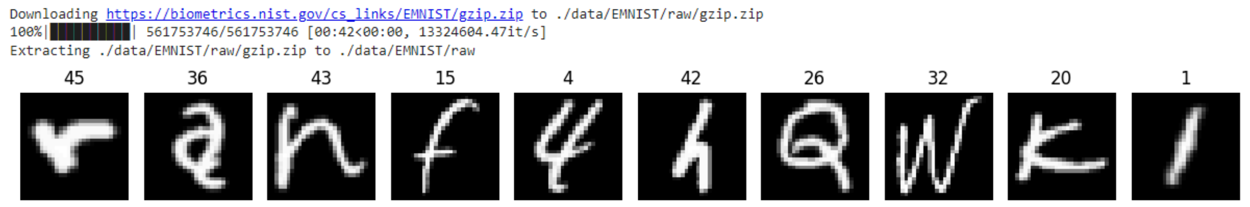
```python
### --YOUR CODE HERE-- ###
ds_val_take10 = ds_val.take(10)

fig, axes = plt.subplots(10, 3, figsize=(15, 40))

for i, (lr_img, hr_img) in enumerate(ds_val_take10):
    gen_img = srgan_generator.predict(tf.expand_dims(lr_img, axis=0))[0]
    axes[i, 0].imshow(lr_img)
    axes[i, 0].set_title('Low-Resolution Image')
    axes[i, 0].axis('off')

    axes[i, 1].imshow(gen_img)
    axes[i, 1].set_title('Generated High-Resolution Image')
    axes[i, 1].axis('off')

    axes[i, 2].imshow(hr_img)
    axes[i, 2].set_title('Original High-Resolution Image')
    axes[i, 2].axis('off')

plt.tight_layout()
plt.show()
```

Figure 13. Code Snippet for Displaying Low-Resolution, Generated, and High-Resolution Images

The code snippet in Figure 13 shows the code for displaying images. The first step is to get 10 low-resolution original images, 10 high-resolution generated images, and 10 high-resolution original images. Once all images are obtained, the images are then plotted for comparison where the low-resolution images are placed in the first column, the high-resolution generated images in the 2nd column, and the high-resolution original images in the 3rd column.

Results and Analysis

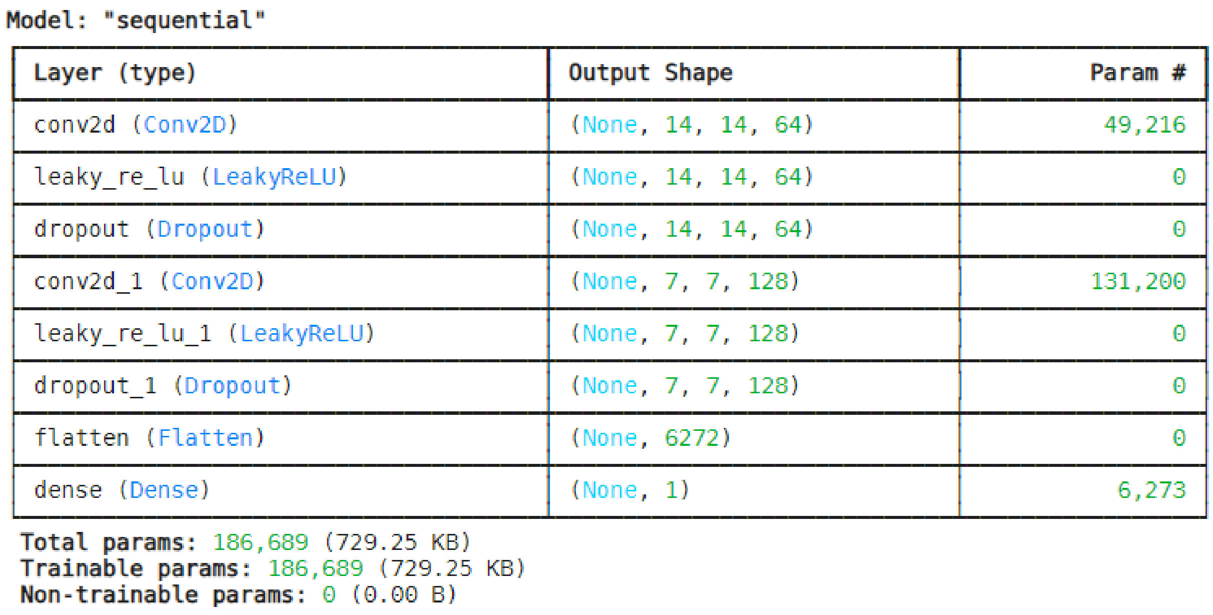
A. Controllable Image Generation using Conditional GAN



```
Downloading https://biometrics.nist.gov/cs_links/EMNIST/gzip.zip to ./data/EMNIST/raw/gzip.zip
100%|██████████| 561753746/561753746 [00:42<00:00, 13324604.47it/s]
Extracting ./data/EMNIST/raw/gzip.zip to ./data/EMNIST/raw
```

Figure 14. EMNIST training dataset samples with their labels

The first part of the activity deals with the importation and assignment of variables for the EMNIST dataset that was used for training and testing the Conditional GAN as shown in Figure 14. This provides the baseline or standard for what to expect for the output of the model. This would mean that a qualitative analysis of the model's output shall take place to test and assess the effectiveness of the model in generating new data.



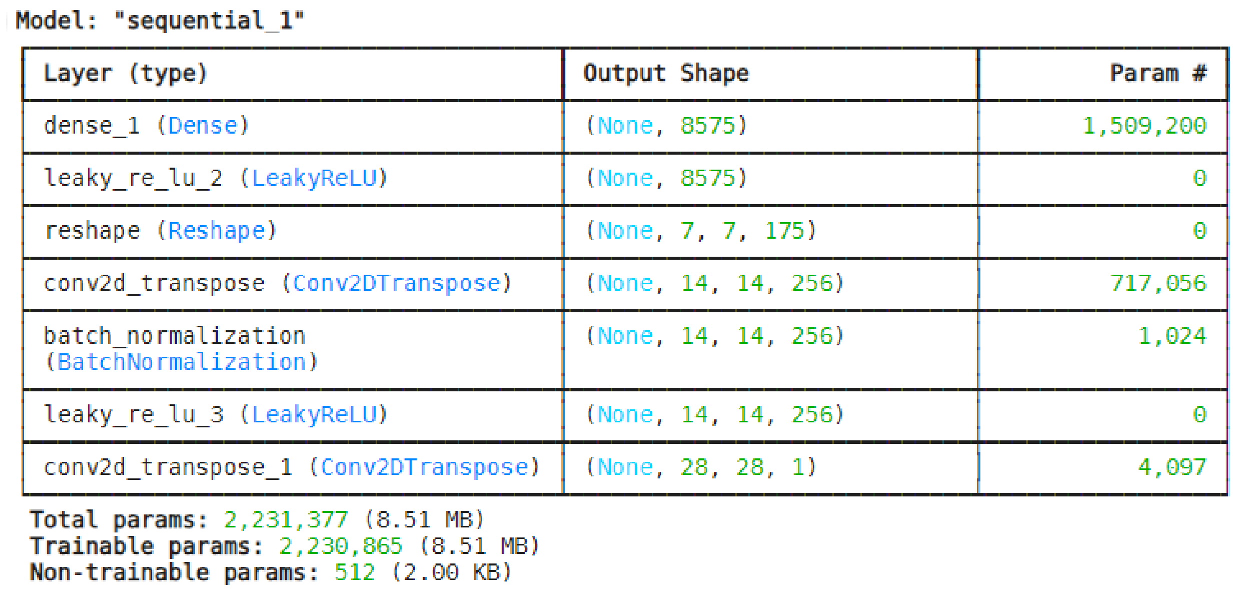
Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 14, 14, 64)	49,216
leaky_re_lu (LeakyReLU)	(None, 14, 14, 64)	0
dropout (Dropout)	(None, 14, 14, 64)	0
conv2d_1 (Conv2D)	(None, 7, 7, 128)	131,200
leaky_re_lu_1 (LeakyReLU)	(None, 7, 7, 128)	0
dropout_1 (Dropout)	(None, 7, 7, 128)	0
flatten (Flatten)	(None, 6272)	0
dense (Dense)	(None, 1)	6,273

```
Total params: 186,689 (729.25 KB)
Trainable params: 186,689 (729.25 KB)
Non-trainable params: 0 (0.00 B)
```

Figure 15. Model Summary of Created Discriminator Subnetwork

The figure above shows the model summary of the created discriminator subnetwork, which matches the model architecture given in the laboratory manual. It shows that there are 186,689 total parameters, all of which are trainable.



```
Model: "sequential_1"
┌──────────────────────────────────────┬─────────────────────┬─────────────┐
│ Layer (type)                         │ Output Shape        │     Param # │
├──────────────────────────────────────┼─────────────────────┼─────────────┤
│ dense_1 (Dense)                      │ (None, 8575)        │   1,509,200 │
├──────────────────────────────────────┼─────────────────────┼─────────────┤
│ leaky_re_lu_2 (LeakyReLU)            │ (None, 8575)        │           0 │
├──────────────────────────────────────┼─────────────────────┼─────────────┤
│ reshape (Reshape)                    │ (None, 7, 7, 175)   │           0 │
├──────────────────────────────────────┼─────────────────────┼─────────────┤
│ conv2d_transpose (Conv2DTranspose)   │ (None, 14, 14, 256) │     717,056 │
├──────────────────────────────────────┼─────────────────────┼─────────────┤
│ batch_normalization                  │ (None, 14, 14, 256) │       1,024 │
│ (BatchNormalization)                 │                     │             │
├──────────────────────────────────────┼─────────────────────┼─────────────┤
│ leaky_re_lu_3 (LeakyReLU)            │ (None, 14, 14, 256) │           0 │
├──────────────────────────────────────┼─────────────────────┼─────────────┤
│ conv2d_transpose_1 (Conv2DTranspose) │ (None, 28, 28, 1)   │       4,097 │
└──────────────────────────────────────┴─────────────────────┴─────────────┘
 Total params: 2,231,377 (8.51 MB)
 Trainable params: 2,230,865 (8.51 MB)
 Non-trainable params: 512 (2.00 KB)
```

Figure 16. Model Summary of Created Generator Subnetwork

The figure shown above displays the model summary of the created generator subnetwork, which matches the provided model architecture in the laboratory manual. Compared to the discriminator, it has more parameters, which makes it more complex, and there are non-trainable parameters that force the subnetwork to learn more effectively. That complexity is necessary for creating high-quality output, and it challenges the discriminator even more.

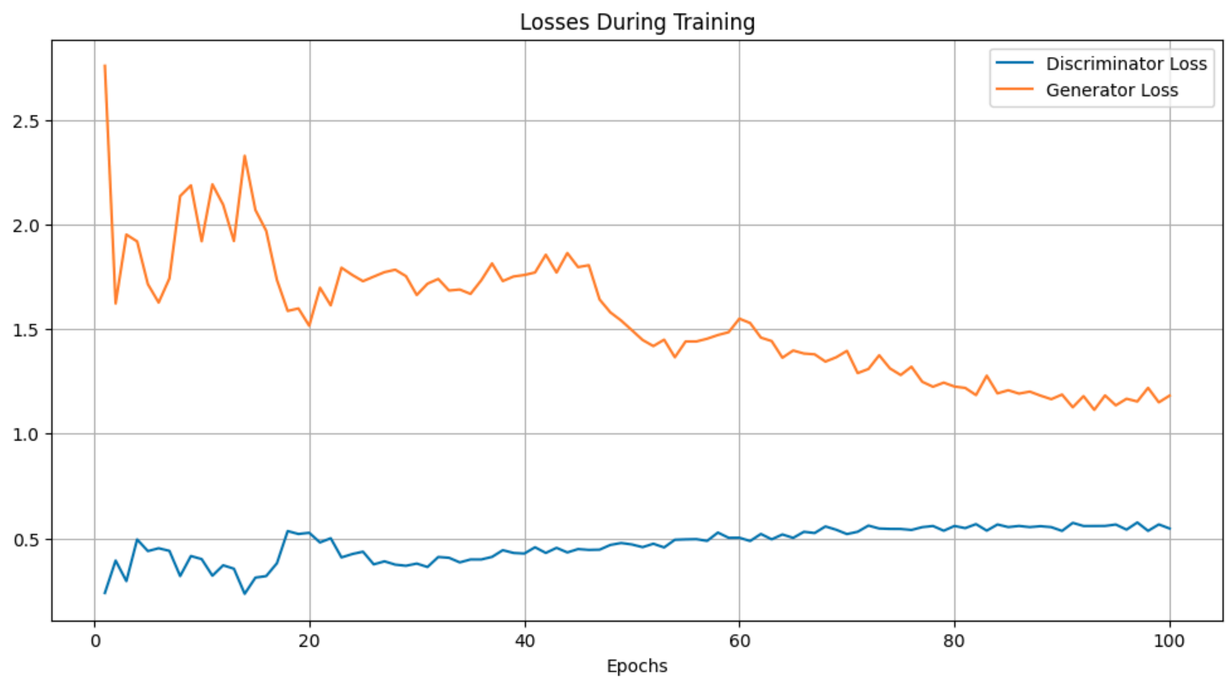


Figure 17. Discriminator and Generator losses graph

Figure 17 shows the loss graphs between the discriminator and generator. The graph can be interpreted as the epochs went by; the generator got better at producing realistic images over time, but at the beginning of the graph, erratic or somewhat random loss was observed, this may be due to the complexity of the task of the generator model and its tendency to find a suitable gradient to work. With that said, the graph did not show a significant rise in the discriminator loss, which could indicate that the discriminator can consistently tell fake images apart generated by the generator and that the value of generator loss did not go below 1, those could be attributed to the just relatively few epochs of training of the GAN model.

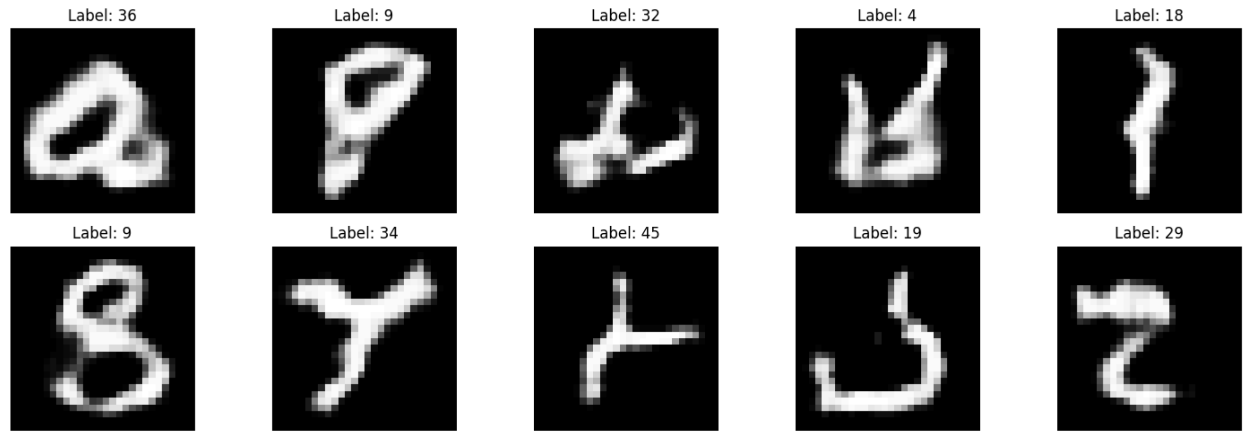


Figure 18. Generated images of the model

The images shown in Figure 18 display the sample outputs of the generator, which ought to be analyzed qualitatively. Although some of the generated images correspond to their actual counterparts, like the images with labels 36, 18, and 34, the rest look just like scribbles or are way different from what they are supposed to be. It is quite apparent that the GAN model still has room for improvement in terms of its generator output, which can be directly attributed to relatively few epochs of training due to the limitations of the Google Colab hardware. However, the model can be improved further by introducing normalization and implementing label smoothing when training the model.

B. Image Super-Resolution using SRGAN

The second part of the activity introduces SRGAN or Super-Resolution Generative Adversarial Network to improve and generate from low-resolution images to high-resolution images. DIV2K dataset with bicubic_x8 data was used in this section where the dataset consists of random low-resolution images. The image below shows the display sample of ten images of the dataset that will be used.

Figure 19. Display Sample of DIV2K Dataset

The generator network architecture was then constructed. The output below shows the summary of the generator network. The input shape of the network is (64, 64, 3), where 64x64 is the image dimension and 3 represents the RGB channels. The first 2 convolution blocks are Conv2D and a single PReLU activation function to introduce the non-linearity of the model. The residual blocks consist of 2D convolution layers, batch normalization, and a PReLU. These residual blocks are repeated 4 times, followed by another 2D convolutional layer and a batch normalization. These are then added from the previous PReLU block. These are followed by 2 upscale blocks, which consist of a 2D convolutional layer and an UpSampling2D.

```
Model: "generator"
__________________________________________________________________________________________________
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================================
 input_layer_1 (InputLayer)  [(None, 64, 64, 3)]          0         []

 conv2d (Conv2D)             (None, 64, 64, 64)           1792      ['input_layer_1[0][0]']

 PReLU_1 (PReLU)             (None, 64, 64, 64)           64        ['conv2d[0][0]']

 conv2d_1 (Conv2D)           (None, 64, 64, 64)           36928     ['PReLU_1[0][0]']

 batch_normalization (Batch  (None, 64, 64, 64)           256       ['conv2d_1[0][0]']
 Normalization)

 p_re_lu (PReLU)             (None, 64, 64, 64)           64        ['batch_normalization[0][0]']

 conv2d_2 (Conv2D)           (None, 64, 64, 64)           36928     ['p_re_lu[0][0]']

 batch_normalization_1 (Bat  (None, 64, 64, 64)           256       ['conv2d_2[0][0]']
 chNormalization)

 add (Add)                   (None, 64, 64, 64)           0         ['PReLU_1[0][0]',
                                                                     'batch_normalization_1[0][0]'
                                                                    ]

 conv2d_3 (Conv2D)           (None, 64, 64, 64)           36928     ['add[0][0]']

 batch_normalization_2 (Bat  (None, 64, 64, 64)           256       ['conv2d_3[0][0]']
 chNormalization)

 p_re_lu_1 (PReLU)           (None, 64, 64, 64)           64        ['batch_normalization_2[0][0]'
                                                                    ]

 conv2d_4 (Conv2D)           (None, 64, 64, 64)           36928     ['p_re_lu_1[0][0]']

 batch_normalization_3 (Bat  (None, 64, 64, 64)           256       ['conv2d_4[0][0]']
 chNormalization)

 add_1 (Add)              (None, 64, 64, 64)         0        ['add[0][0]',
                                                                     'batch_normalization_3[0][0]'
                                                                    ]

 conv2d_5 (Conv2D)        (None, 64, 64, 64)         36928    ['add_1[0][0]']

 batch_normalization_4 (Bat  (None, 64, 64, 64)      256      ['conv2d_5[0][0]']
 chNormalization)

 p_re_lu_2 (PReLU)        (None, 64, 64, 64)         64       ['batch_normalization_4[0][0]'
                                                                    ]

 conv2d_6 (Conv2D)        (None, 64, 64, 64)         36928    ['p_re_lu_2[0][0]']

 batch_normalization_5 (Bat  (None, 64, 64, 64)      256      ['conv2d_6[0][0]']
 chNormalization)

 add_2 (Add)              (None, 64, 64, 64)         0        ['add_1[0][0]',
                                                                     'batch_normalization_5[0][0]'
                                                                    ]

 conv2d_7 (Conv2D)        (None, 64, 64, 64)         36928    ['add_2[0][0]']

 batch_normalization_6 (Bat  (None, 64, 64, 64)      256      ['conv2d_7[0][0]']
 chNormalization)

 p_re_lu_3 (PReLU)        (None, 64, 64, 64)         64       ['batch_normalization_6[0][0]'
                                                                    ]

 conv2d_8 (Conv2D)        (None, 64, 64, 64)         36928    ['p_re_lu_3[0][0]']

 batch_normalization_7 (Bat  (None, 64, 64, 64)      256      ['conv2d_8[0][0]']
 chNormalization)

 add_3 (Add)              (None, 64, 64, 64)         0        ['add_2[0][0]',
                                                                     'batch_normalization_7[0][0]'
                                                                    ]

 conv2d_22 (Conv2D)       (None, 64, 64, 64)         36928    ['add_3[0][0]']

 batch_normalization_22 (Ba  (None, 64, 64, 64)      256      ['conv2d_22[0][0]']
 tchNormalization)

 add_4 (Add)              (None, 64, 64, 64)         0        ['batch_normalization_22[0][0]
                                                                    ',
                                                  'PReLU_1[0][0]']

  conv2d_9 (Conv2D)          (None, 64, 64, 256)          147712    ['add_4[0][0]']

  up_sampling2d (UpSampling2  (None, 128, 128, 256)       0         ['conv2d_9[0][0]']
  D)

  p_re_lu_4 (PReLU)          (None, 128, 128, 256)       256       ['up_sampling2d[0][0]']

  conv2d_10 (Conv2D)         (None, 128, 128, 256)       590080    ['p_re_lu_4[0][0]']

  up_sampling2d_1 (UpSamplin  (None, 256, 256, 256)       0         ['conv2d_10[0][0]']
  g2D)

  p_re_lu_5 (PReLU)          (None, 256, 256, 256)       256       ['up_sampling2d_1[0][0]']

  conv2d_25 (Conv2D)         (None, 256, 256, 3)         6915      ['p_re_lu_5[0][0]']

 ==================================================================================================
 Total params: 1081987 (4.13 MB)
 Trainable params: 1080835 (4.12 MB)
 Non-trainable params: 1152 (4.50 KB)
 __________________________________________________________________________________________________
```

Table 1. Output Summary of the Generator Network

The discriminator block, on the other hand, has an input shape of (256, 256, 3), where 256x256 is the image dimension and 3 represents the RGB channels. It has a series of convolutional layers that progressively reduce the spatial dimensions. Each convolutional layer is followed by a Leaky ReLU activation function and batch normalization. The output of the final convolutional layer is flattened into a 1D vector by the flattened layer, which is then passed through a dense layer with 1024 units and a Leaky ReLU activation. Lastly, the final dense layer outputs a single value, which represents the probability of the input images, whether they are fake or real.

```
Model: "discriminator"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer_1 (InputLayer)  [(None, 256, 256, 3)]     0

 conv2d_37 (Conv2D)          (None, 256, 256, 64)      1792

 leaky_re_lu (LeakyReLU)     (None, 256, 256, 64)      0
```

```
conv2d_11 (Conv2D)         (None, 128, 128, 64)     36928

batch_normalization_8 (Bat  (None, 128, 128, 64)     256
chNormalization)

leaky_re_lu_1 (LeakyReLU)   (None, 128, 128, 64)     0

conv2d_12 (Conv2D)         (None, 128, 128, 128)    73856

batch_normalization_9 (Bat  (None, 128, 128, 128)    512
chNormalization)

leaky_re_lu_2 (LeakyReLU)   (None, 128, 128, 128)    0

conv2d_13 (Conv2D)         (None, 64, 64, 128)      147584

batch_normalization_10 (Ba  (None, 64, 64, 128)      512
tchNormalization)

leaky_re_lu_3 (LeakyReLU)   (None, 64, 64, 128)      0

conv2d_14 (Conv2D)         (None, 64, 64, 256)      295168

batch_normalization_11 (Ba  (None, 64, 64, 256)      1024
tchNormalization)

leaky_re_lu_4 (LeakyReLU)   (None, 64, 64, 256)      0

conv2d_15 (Conv2D)         (None, 32, 32, 256)      590080

batch_normalization_12 (Ba  (None, 32, 32, 256)      1024
tchNormalization)

leaky_re_lu_5 (LeakyReLU)   (None, 32, 32, 256)      0

conv2d_16 (Conv2D)         (None, 32, 32, 512)      1180160

batch_normalization_13 (Ba  (None, 32, 32, 512)      2048
tchNormalization)

leaky_re_lu_6 (LeakyReLU)   (None, 32, 32, 512)      0

flatten (Flatten)          (None, 524288)           0

dense (Dense)              (None, 1024)             536871936

leaky_re_lu_7 (LeakyReLU)   (None, 1024)             0
```

```
 dense_1 (Dense)          (None, 1)              1025

=================================================================
Total params: 539203905 (2.01 GB)
Trainable params: 539201217 (2.01 GB)
Non-trainable params: 2688 (10.50 KB)
_________________________________________________________________
```

Table 2. Output Summary of Discriminator Network

Next is to set up the SRGAN using the pre-trained VGG19 model from Keras. The VGG19 model is loaded with weights from the ImageNet dataset, excluding its top layers, which are used to extract important features from high-resolution images. The SRGAN model inputs the low-resolution image and uses the generator to generate a high-resolution image from the input to convert it to a high-resolution version, while the discriminator determines whether the generated image is fake or not when compared to the high-resolution image. The final output of the SRGAN is the evaluation of the discriminator which can help to create a clearer and more detailed image.

The SRGAN is then trained with binary cross entropy as its loss function and Adam as its optimizer. The model is trained with a batch size and epochs of 10. The training took an approximate time of 2 hours, considering the limited amount of resources available in Google Colab. TPU v2-8 was used for the hardware accelerator for the runtime type. As seen in the figure below, the losses in the first few epochs of the generator start high and eventually lower until it reaches around 0.35. This indicates that the generator gradually improves as epochs increase. The discriminator on the other hand, shows an almost constant loss close to 0.69.

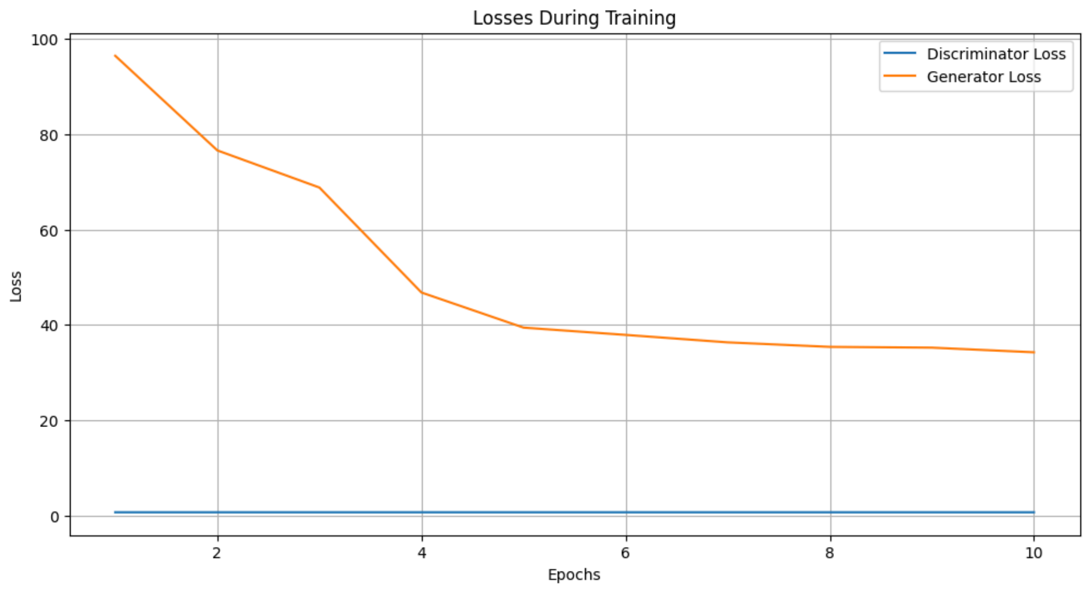
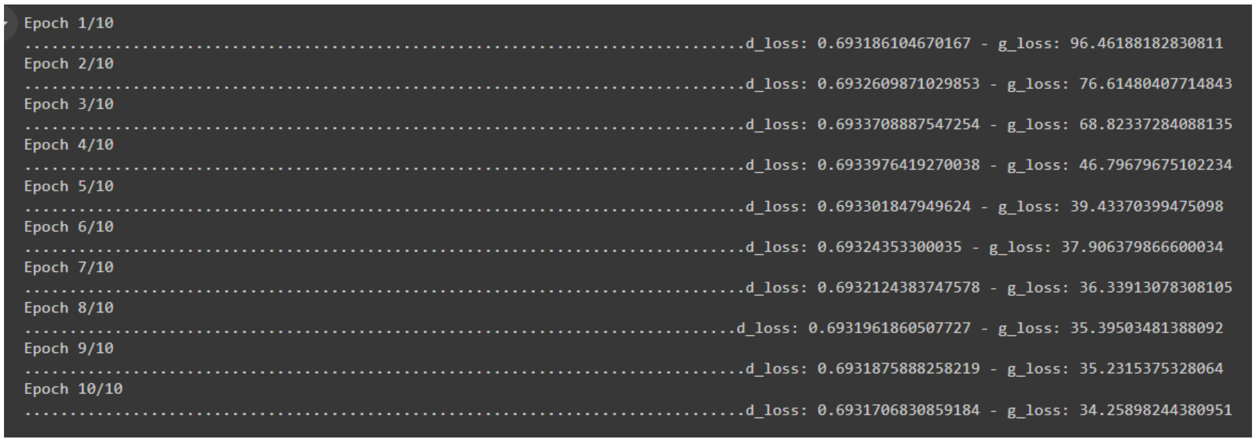


Figure 20. Losses of Generator and Discriminator Networks During Training



```
Epoch 1/10
..............................................................................d_loss: 0.693186104670167 - g_loss: 96.46188182830811
Epoch 2/10
..............................................................................d_loss: 0.6932609871029853 - g_loss: 76.61480407714843
Epoch 3/10
..............................................................................d_loss: 0.6933708887547254 - g_loss: 68.82337284088135
Epoch 4/10
..............................................................................d_loss: 0.6933976419270038 - g_loss: 46.79679675102234
Epoch 5/10
..............................................................................d_loss: 0.693301847949624 - g_loss: 39.43370399475098
Epoch 6/10
..............................................................................d_loss: 0.69324353300035 - g_loss: 37.906379866600034
Epoch 7/10
..............................................................................d_loss: 0.6932124383747578 - g_loss: 36.33913078308105
Epoch 8/10
.............................................................................d_loss: 0.6931961860507727 - g_loss: 35.39503481388092
Epoch 9/10
..............................................................................d_loss: 0.6931875888258219 - g_loss: 35.2315375328064
Epoch 10/10
..............................................................................d_loss: 0.6931706830859184 - g_loss: 34.25898244380951
```

Figure 21. Looking further into Discriminator Loss

It can be seen in Figure 20 that the discriminator loss remains relatively constant, with little difference from the first until the last epoch. In other GAN models, the expected behavior of the discriminator loss is the mirror opposite of the generator loss. This behavior of having a constant discriminator loss may be caused by using VGG19 in the generator, which already has pre-trained weights that give the generator a significant advantage from the beginning, making it harder for the discriminator to distinguish between real and generated images. The group also observed that the discriminator loss value is usually around 0.693, as shown in Figure 21, which is equivalent to ln(2) (natural logarithm of 2). This means that the discriminator is giving an output probability close to 0.5, indicating that it thinks that the image is both 50% fake and 50% real at the same time, also indicating maximum entropy. This uncertainty may also be caused by the output resolution being in the middle ground between the original high-resolution image and the low-resolution image. Another cause for the discriminator loss behavior is due to lack of training of the discriminator for real high-resolution images. At the very beginning of the training, before training the generator, the discriminator weights could be unfrozen and could be instead trained on high-resolution original images so that the discriminator can more confidently distinguish between real high-resolution and fake high-resolution images in the succeeding batches.

The output results of the SRGAN model are shown in the figures below. The output shows comparable images of the low-resolution input image, the generated image, and the high-resolution images. All ten output images have resembled the original images. The generated high-resolution image showed an excellent result, but it is less sharp than the original high-resolution image. Despite this and the unexpected constant behavior of the discriminator loss, the generated image is still satisfactory, offering a clearer quality compared to the initial low-resolution input image.

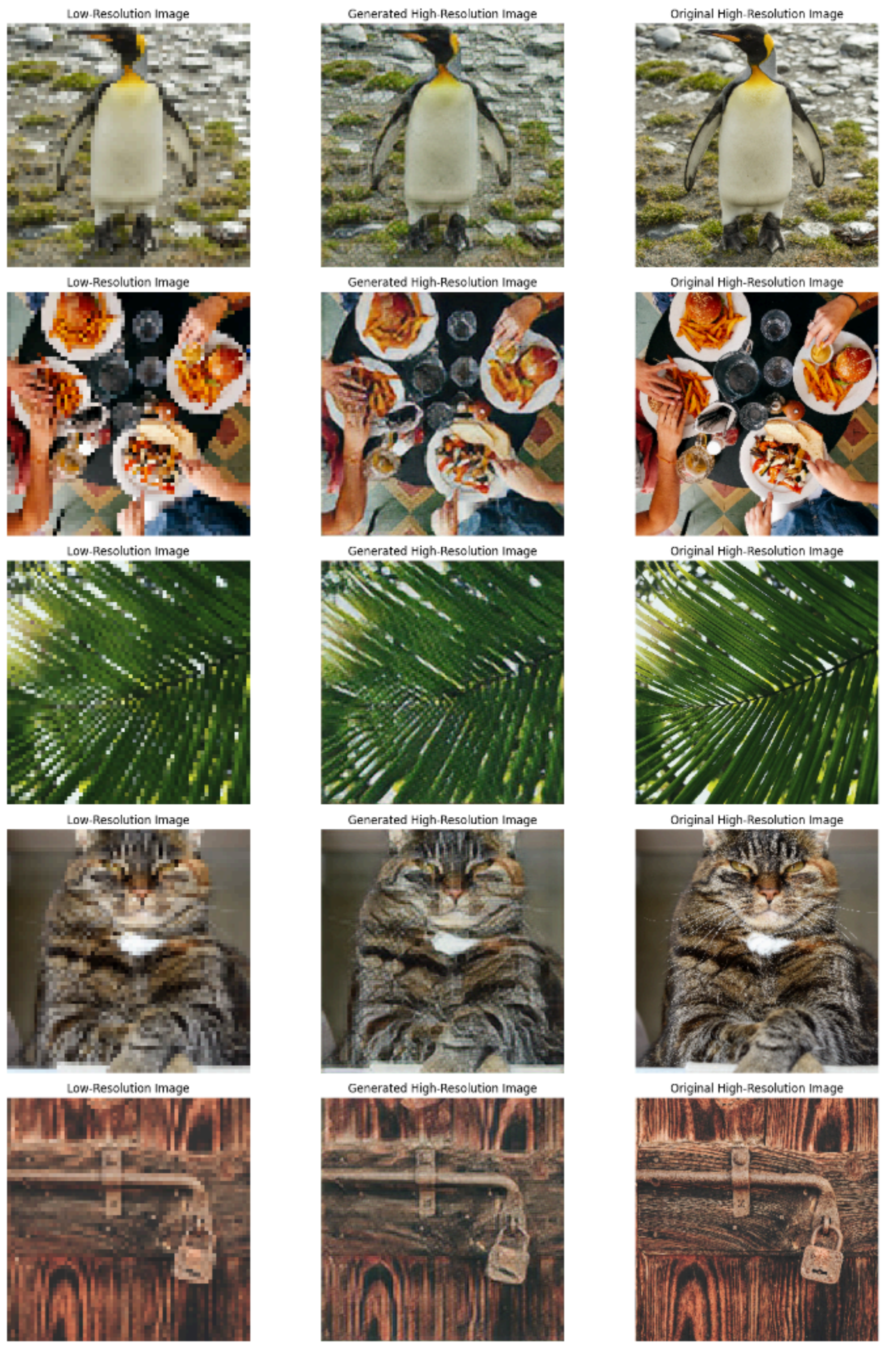

Low-Resolution Image
Generated High-Resolution Image
Original High-Resolution Image
Low-Resolution Image
Generated High-Resolution Image
Original High-Resolution Image
Low-Resolution Image
Generated High-Resolution Image
Original High-Resolution Image
Low-Resolution Image
Generated High-Resolution Image
Original High-Resolution Image
Low-Resolution Image
Generated High-Resolution Image
Original High-Resolution Image

Figure 22. Output Images 1

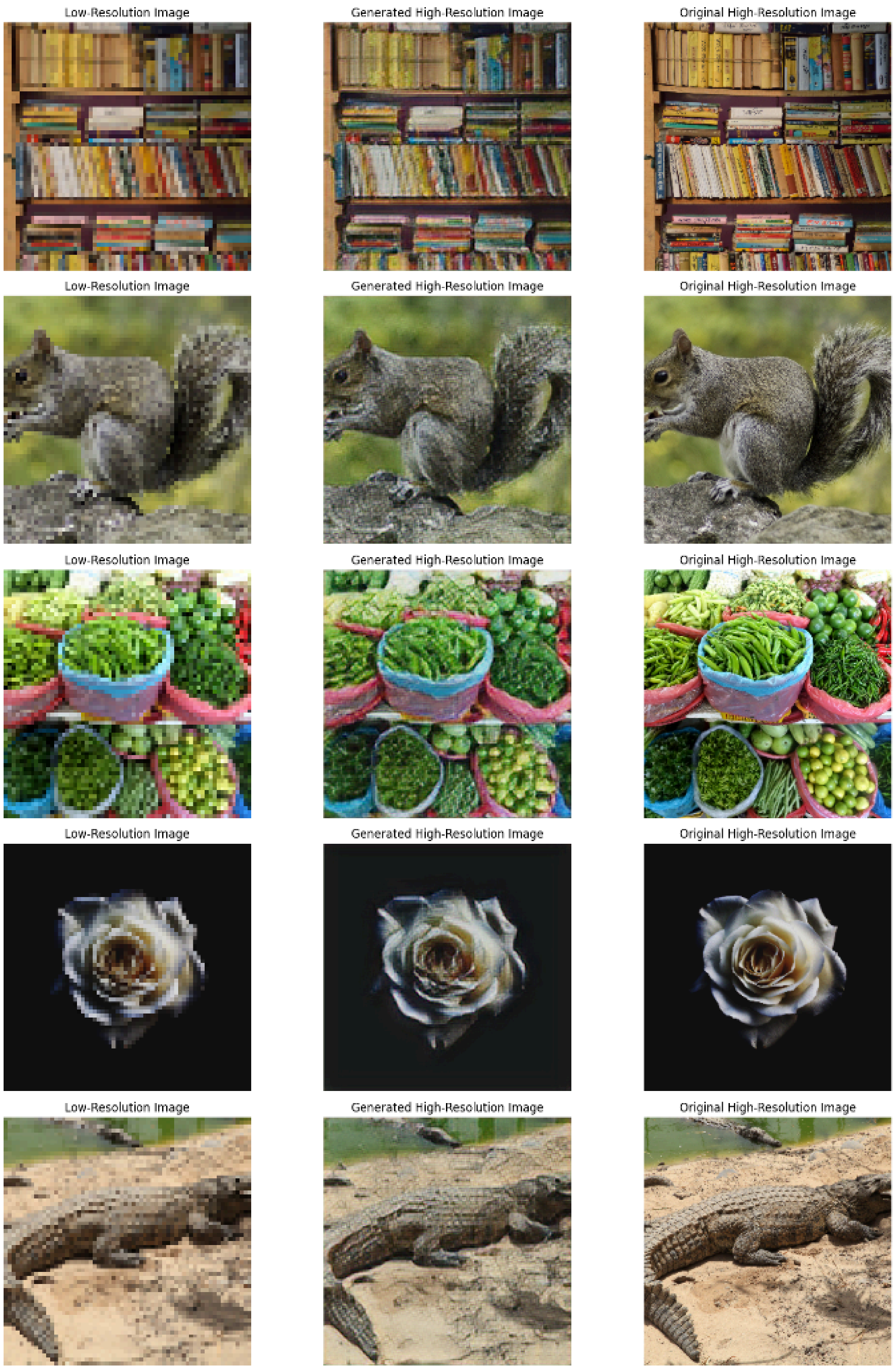

Low-Resolution Image
Generated High-Resolution Image
Original High-Resolution Image

Figure 23. Output Images 2

Conclusions

The third laboratory activity introduces the application of image-generative models using GAN models that use machine learning and deep learning to create a realistic, high-quality image that mimics real-world images. The first part of the activity uses the EMNIST dataset to generate handwritten images using conditional GAN. The output shows relatively sharp but some produced a random scribble that does not resemble any character. This could be improved when higher epochs are done in training, but due to the constraints of time and limited resources of Google Colab, the students were not able to train at a higher epoch count.

The second part of the activity uses DIV2K Dataset bicubic_x8 data by using SRGAN or Super-Resolution Generative Adversarial Network to generate higher-resolution images from low-resolution input. By building the generator and discriminator network with the purpose of enhancing image quality, the results show that the SRGAN model was able to improve and rebuild the low-resolution pixelated image. The generated high-resolution image, shown in the results, obtained an excellent result, but when compared to the original high-resolution image, the generated image has a lower sharpness output. Nonetheless, despite the unexpected behavior of the discriminator, the generated image has a satisfactory result, which indeed obtained a clearer image quality compared to the low-resolution image input.

Overall, this laboratory activity has helped the students understand the application of image generator models using GAN. Though the training took a large amount of time and limited resources, the students were able to partially accomplish the activity. The students also gained hands-on experience in training and fine-tuning GANs, which has improved their understanding of the applications of deep learning. This activity will be useful in future generative AI application activities.

References

Chakraborty, D. (2022, July 14). *Super-Resolution Generative Adversarial Networks (SRGAN) - PyImageSearch*. PyImageSearch. https://pyimagesearch.com/2022/06/06/super-resolution-generative-adversarial-networks-srgan/

GeeksforGeeks. (2024, July 17). *Conditional GANs (CGANs) for image generation*. GeeksforGeeks. https://www.geeksforgeeks.org/conditional-gans-cgans-for-image-generation/