Imports

```
# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt
import datetime
tf.keras.backend.clear_session()
import glob
import matplotlib.pyplot as plt
import PIL
import os
from PIL import Image
import time
from tqdm import tqdm
from IPython import display
import warnings
warnings.filterwarnings("ignore")
```

```
!unzip /content/archive.zip
```

Extracting filenames

```
file_name = list()
for filename in tqdm(glob.glob('/content/img_align_celeba
  file_name.append(filename)
```

```
file_name.append(filename)
```

```
100%|██████████| 202599/202599 [00:00<00:00, 295430
```

Limiting the dataset to 2000 images for faster training

```
file_path= file_name[0:20000]
```

Croping the image to capture the face of celebs in each image

```
crop = (30, 55, 150, 175)
images = [np.array((Image.open(path).crop(crop)).resize(
images = np.array(images)
images.shape
```

Normalizing the images between [-1, 1]

```
train_images = images.reshape(images.shape[0], 64, 64, 3
train_images = (train_images -127.5) / 127.5
train_images.shape
```

Convert train_images into a tf.data.Dataset

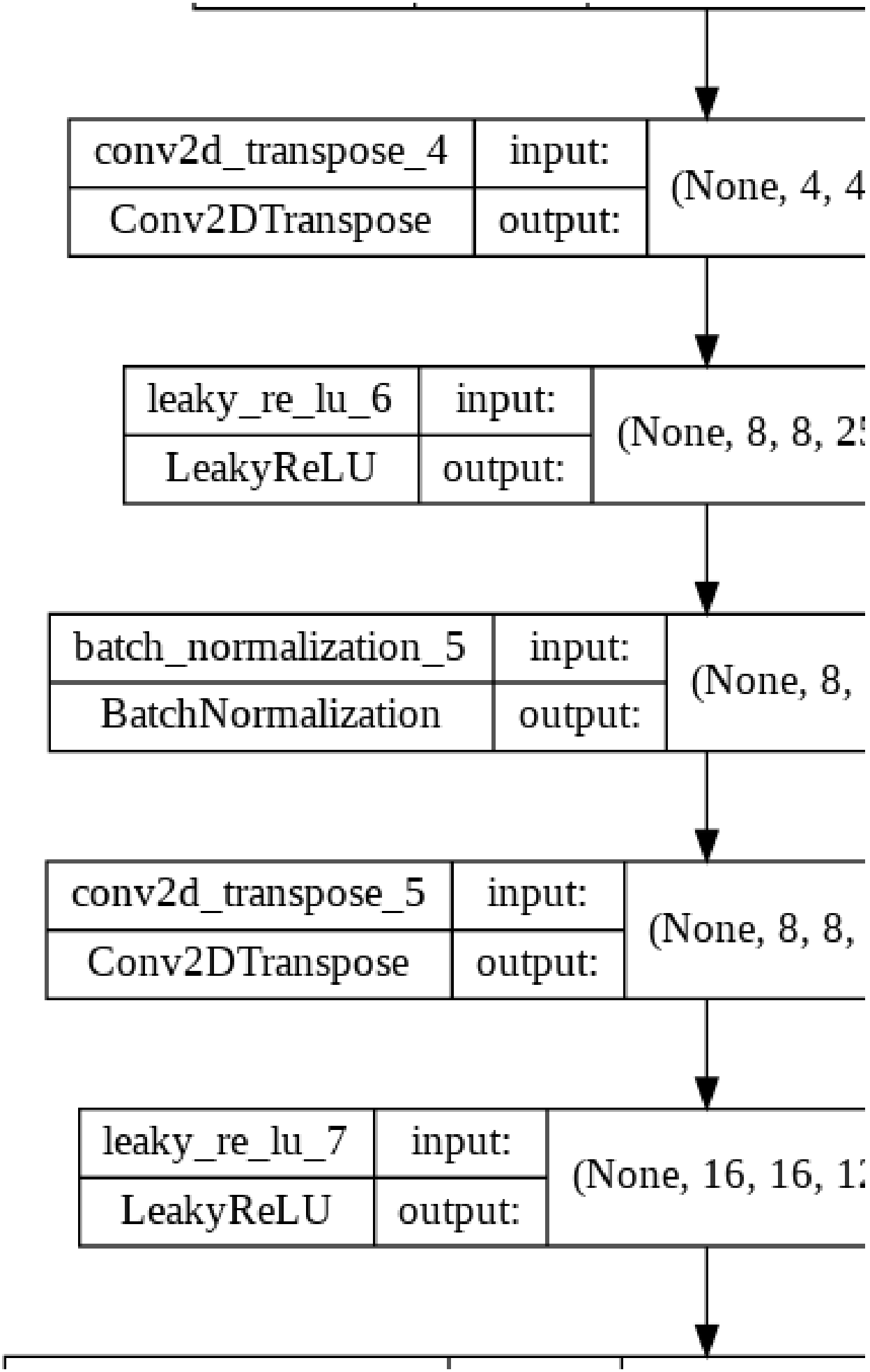
```
BUFFER_SIZE = 20000
BATCH_SIZE = 256

# Batch and shuffle the data
# Refer: https://www.tensorflow.org/api_docs/python/tf/da
'''
Creates a Dataset whose elements are slices of the given
The given tensors are sliced along their first dimension.
'''
train_dataset = tf.data.Dataset.from_tensor_slices(train_
print(train_dataset)
```

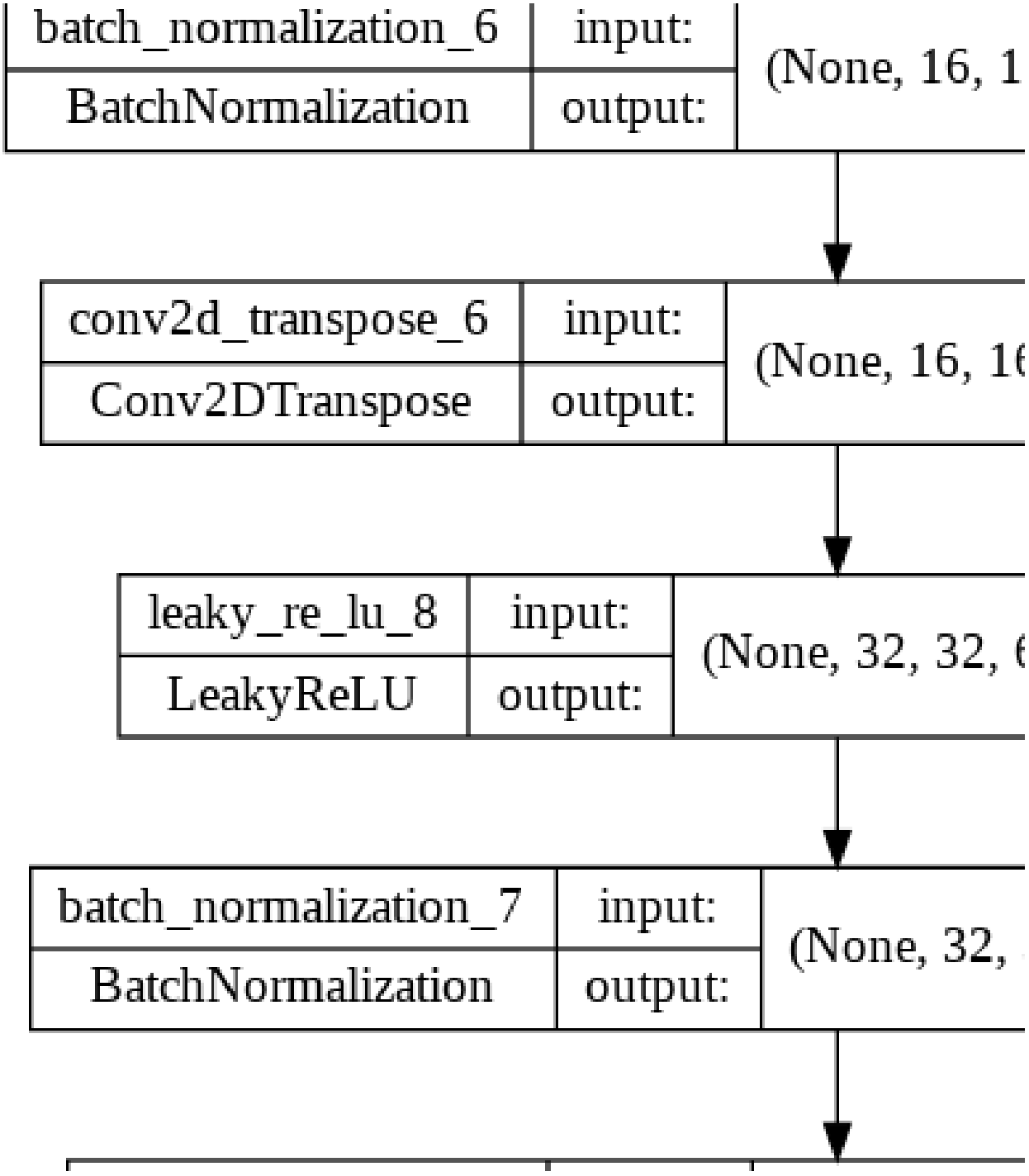
```
<BatchDataset element_spec=TensorSpec(shape=(None,
```

Generator Model

```
def make_generator_model():
    generator=tf.keras.Sequential()

    generator.add(layers.Dense(4*4*512,input_shape=[100])
    generator.add(layers.Reshape([4,4,512]))

    generator.add(layers.Conv2DTranspose(256, kernel_size
    generator.add(layers.LeakyReLU(alpha=0.2))
    generator.add(layers.BatchNormalization())

    generator.add(layers.Conv2DTranspose(128, kernel_size
    generator.add(layers.LeakyReLU(alpha=0.2))
    generator.add(layers.BatchNormalization())
```

```
    generator.add(layers.Conv2DTranspose(64, kernel_size=
    generator.add(layers.LeakyReLU(alpha=0.2))
    generator.add(layers.BatchNormalization())

    generator.add(layers.Conv2DTranspose(3, kernel_size=4
                                         activation='tanh'))
    return generator


generator = make_generator_model()
keras.utils.plot_model(generator, 'generator.png', show_s
```

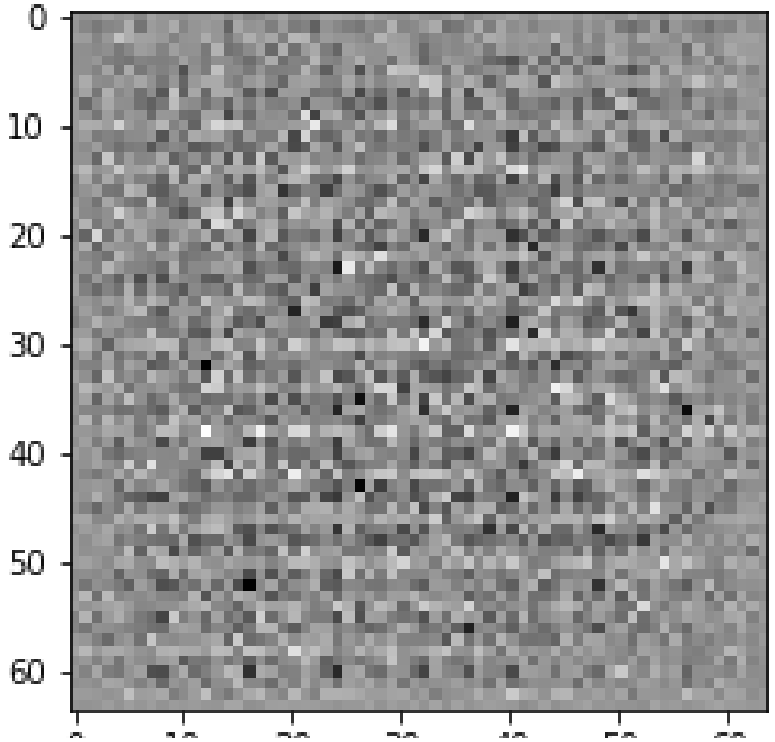

conv2d_transpose_4
Conv2DTranspose
input:
output:
(None, 4, 4
leaky_re_lu_6
LeakyReLU
input:
output:
(None, 8, 8, 2
batch_normalization_5
BatchNormalization
input:
output:
(None, 8,
conv2d_transpose_5
Conv2DTranspose
input:
output:
(None, 8, 8,
leaky_re_lu_7
LeakyReLU
input:
output:
(None, 16, 16, 1



Forward-pass through the generator

```
noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)
plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

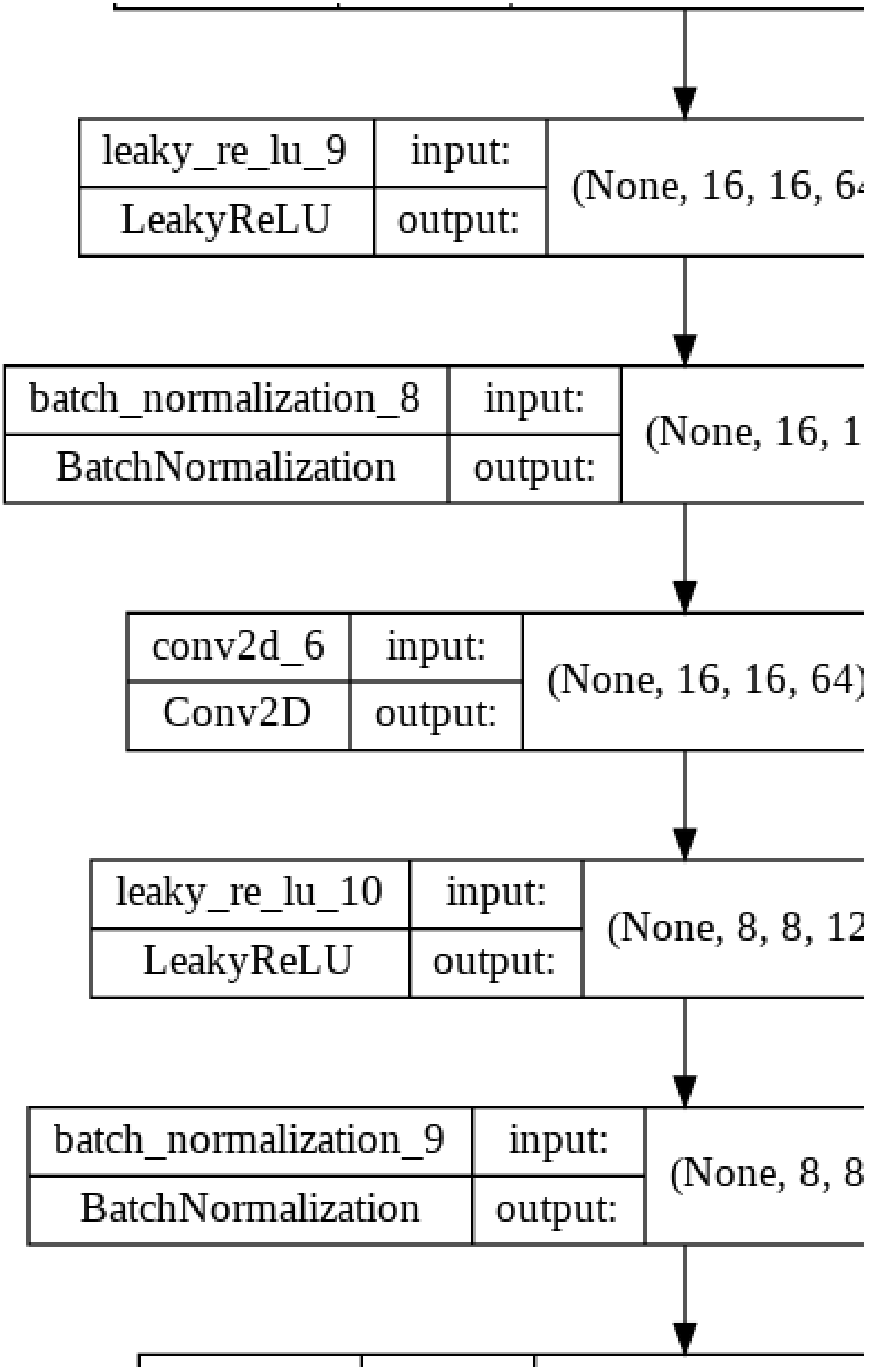
```
<matplotlib.image.AxesImage at 0x7ff5c302f510>
```

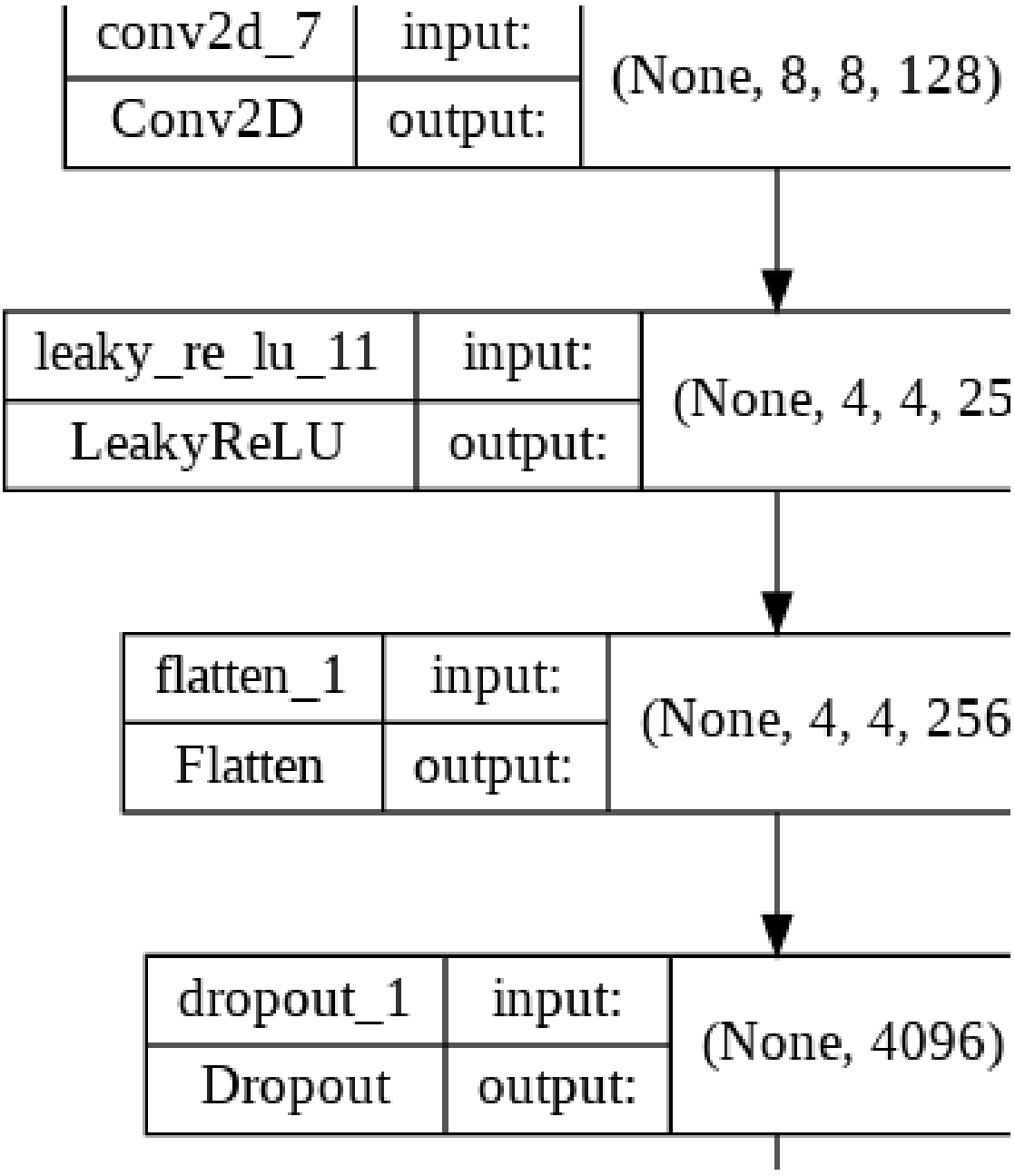


Discriminator Model

```
def make_discriminator_model():
  discriminator= tf.keras.Sequential()
  discriminator.add(layers.Conv2D(32, kernel_size=4, stri

  discriminator.add(layers.Conv2D(64, kernel_size=4, stri
  discriminator.add(layers.LeakyReLU(0.2))
  discriminator.add(layers.BatchNormalization())

  discriminator.add(layers.Conv2D(128, kernel_size=4, str
  discriminator.add(layers.LeakyReLU(0.2))
  discriminator.add(layers.BatchNormalization())

  discriminator.add(layers.Conv2D(256, kernel_size=4, str
  discriminator.add(layers.LeakyReLU(0.2))
```

```
    discriminator.add(layers.Flatten())
    discriminator.add(layers.Dropout(0.5))
    discriminator.add(layers.Dense(1))

    return discriminator



discriminator = make_discriminator_model()
keras.utils.plot_model(discriminator, 'discriminator.png
```


leaky_re_lu_9
input:
LeakyReLU
output:
(None, 16, 16, 6
batch_normalization_8
input:
BatchNormalization
output:
(None, 16, 1
conv2d_6
input:
Conv2D
output:
(None, 16, 16, 64)
leaky_re_lu_10
input:
LeakyReLU
output:
(None, 8, 8, 12
batch_normalization_9
input:
BatchNormalization
output:
(None, 8, 8



Forward-pass through discriminator model, not trained yet

| dense_3 | input: |

```
decision = discriminator(generated_image)
print (decision)
```

```
tf.Tensor([[-0.03973623]], shape=(1, 1), dtype=floa
```

Loss functions

```
# Loss: Binary CrossEntropy <=> log-loss
# This method returns a helper function to compute cross
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_
```

Discrimanator Loss

```
# For the discrimanator,  0 => fake and 1 => real image
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output),
    fake_loss = cross_entropy(tf.zeros_like(fake_output),
    total_loss = real_loss + fake_loss
    return total_loss
```

Generator Loss

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_
```

Setting Optimizers and Checkpoints

```
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

```
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=gene
                                 discriminator_optimizer=
                                 generator=generator,
                                 discriminator=discrimina
```

Creating 20 samples of random noise from normal distribution

```
EPOCHS = 420
noise_dim = 100
num_examples_to_generate = 20
# We will reuse these test_random_vectors overtime (so it
# to visualize progress in the animated GIF)
test_random_vectors = tf.random.normal([num_examples_to_g
print(test_random_vectors.shape)
```

```
(20, 100)
```

Training utilities

```
# Notice the use of `tf.function`
# This annotation causes the function to be "compiled" in
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])
```

```
    with tf.GradientTape() as gen_tape, tf.GradientTape(
      generated_images = generator(noise, training=True)

      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, trai

      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_ou

    gradients_of_generator = gen_tape.gradient(gen_loss,
    gradients_of_discriminator = disc_tape.gradient(disc_

    generator_optimizer.apply_gradients(zip(gradients_of_
    discriminator_optimizer.apply_gradients(zip(gradients
    return gen_loss, disc_loss



def train(dataset, epochs):
  for epoch in range(epochs):
    start = time.time()

    for image_batch in dataset:
      gen_loss, disc_loss = train_step(image_batch)

    # Produce images for the GIF as we go
    display.clear_output(wait=True)
    generate_and_save_images(generator, epoch + 1, test_

    # Save the model every 15 epochs
    if (epoch + 1) % 15 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)
```

```
    print ('Time for epoch {} is {} sec'.format(epoch + 1
  # Generate after the final epoch
  display.clear_output(wait=True)
  generate_and_save_images(generator, epochs, test_randor
```

Generating and saving the the images from trained model

```
def generate_and_save_images(model, epoch, test_input, ge
  # Notice `training` is set to False.
  # This is so all layers run in inference mode (batchnor
  predictions = model(test_input, training=False)
  fig = plt.figure(figsize=(8, 8))
  for i in range(predictions.shape[0]):
      plt.subplot(4, 5, i+1)
      plt.imshow(tf.cast((predictions[i]* 127.5) + 127.5,
      plt.axis('off')
  print('Generative Model loss: '+ str(gen_loss.numpy()))
  print('Discriminator Model loss: ' + str(disc_loss.nump

  plt.savefig('image3_at_epoch_{:04d}.png'.format(epoch))
  plt.show()


train(train_dataset, EPOCHS)
```