```
import os
import numpy as np
import matplotlib.pyplot as plt
from tqdm.notebook import tqdm
import warnings
from PIL import Image

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import load_img, array_to_img
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras import layers
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import BinaryCrossentropy

warnings.filterwarnings('ignore')
```

```
/opt/conda/lib/python3.10/site-packages/scipy/__init__.py:146: UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required f
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
/opt/conda/lib/python3.10/site-packages/tensorflow_io/python/ops/__init__.py:98: UserWarning: unable to load libtensorflow_io_
caused by: ['/opt/conda/lib/python3.10/site-packages/tensorflow_io/python/ops/libtensorflow_io_plugins.so: undefined symbol: _
  warnings.warn(f"unable to load libtensorflow_io_plugins.so: {e}")
/opt/conda/lib/python3.10/site-packages/tensorflow_io/python/ops/__init__.py:104: UserWarning: file system plugins are not loa
caused by: ['/opt/conda/lib/python3.10/site-packages/tensorflow_io/python/ops/libtensorflow_io.so: undefined symbol: _ZTVN10te
  warnings.warn(f"file system plugins are not loaded: {e}")
```

```
pip install Pillow
```

```
Requirement already satisfied: Pillow in /opt/conda/lib/python3.10/site-packages (9.5.0)
Note: you may need to restart the kernel to use updated packages.
```

## Load Dataset

```
# Get the images folder (You can get this directly like this if you're using Kaggle notebook,
# else you'll have to download it)
input_image = '/kaggle/input/animefacedataset/images'
```

```
image_list = []

# Create an image_list of all image paths
for image in os.listdir(input_image):

    image_list.append(os.path.join(input_image,image))
```

```
image_list[:5] # Peek the top 5 items of the list
```

```
['/kaggle/input/animefacedataset/images/35715_2011.jpg',
 '/kaggle/input/animefacedataset/images/48610_2014.jpg',
 '/kaggle/input/animefacedataset/images/34719_2011.jpg',
 '/kaggle/input/animefacedataset/images/40266_2012.jpg',
 '/kaggle/input/animefacedataset/images/4199_2002.jpg']
```

```
len(image_list)
```

```
63565
```

```
train_images = []

for path in image_list:
    img = Image.open(path) # Loads each image path
    img = img.resize((64,64)) # Resize the image to 64×64
    image = np.array(img) # Convert it to a NumPy array
    train_images.append(image) # Collect array into train_images for model training
```

```
train_images = np.array(train_images) # Convert entire array to numpy array
```

```
# Machine learning models expect the format:
# (number_of_images, height, width, channels)
train_images = train_images.reshape(
    train_images.shape[0],  # number of images
    64,                     # image height
    64,                     # image width
    3                       # channels (RGB)
).astype('float32')         # convert datatype
```

```
# Normalize (0,255) => (-1,1)   (255-127.5)/127.5 = 1 //////  (0-127.5)/127.5 = -1
train_images = (train_images - 127.5)/127.5
```

```
train_images[0] # Peak the first normalized image
```

```
array([[[ 0.28627452,  0.24705882,  0.07450981],
        [ 0.8509804 ,  0.81960785,  0.62352943],
        [ 0.2       ,  0.18431373, -0.04313726],
        ...,
        [ 0.24705882,  0.22352941, -0.09019608],
        [ 0.35686275,  0.3254902 ,  0.05098039],
        [ 0.827451  ,  0.77254903,  0.5529412 ]],

       [[ 1.        ,  1.        ,  0.8039216 ],
        [ 0.34117648,  0.30980393,  0.09803922],
        [-0.07450981, -0.08235294, -0.3254902 ],
        ...,
        [ 0.5058824 ,  0.49019608,  0.11372549],
        [-0.06666667, -0.09803922, -0.39607844],
        [ 0.5058824 ,  0.45882353,  0.20784314]],

       [[ 0.11372549,  0.08235294, -0.14509805],
        [-0.54509807, -0.5686275 , -0.8117647 ],
        [ 0.8509804 ,  0.8352941 ,  0.5372549 ],
        ...,
        [ 0.5294118 ,  0.5137255 ,  0.06666667],
        [ 0.01960784, -0.01176471, -0.38039216],
        [ 0.08235294,  0.04313726, -0.25490198]],

       ...,

       [[ 0.9764706 ,  1.        ,  0.9843137 ],
        [ 0.9529412 ,  0.99215686,  0.94509804],
        [ 0.9607843 ,  1.        ,  0.94509804],
        ...,
        [ 0.8745098 ,  0.84313726,  0.4745098 ],
        [ 0.10588235,  0.09803922, -0.24705882],
        [ 0.23921569,  0.23137255, -0.10588235]],

       [[ 0.8745098 ,  0.90588236,  0.8980392 ],
        [ 0.92156863,  0.9607843 ,  0.92941177],
        [ 0.9372549 ,  0.9764706 ,  0.92156863],
        ...,
        [ 0.8901961 ,  0.85882354,  0.49803922],
        [ 0.25490198,  0.23921569, -0.08235294],
```

```
          [ 0.04313726,  0.04313726, -0.25490198]],

         [[ 0.9764706 ,  1.        ,  1.        ],
          [ 0.9607843 ,  0.99215686,  0.9843137 ],
          [ 0.99215686,  1.        ,  0.9607843 ],
          ...,
          [ 0.8352941 ,  0.8039216 ,  0.4509804 ],
          [ 0.35686275,  0.35686275,  0.05882353],
          [-0.02745098, -0.01960784, -0.28627452]]], dtype=float32)
```

### Create Generator

```
# Latent dimension for random noise
LATENT_DIM = 300
# Weight initializer
WEIGHT_INIT = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)
# Number of channels of the image
CHANNELS = 3 # For gray scale, keep it as 1
```

### Generator Model

```
# Sequential is a simple Keras model class that stacks layers in order, one after another.
model = Sequential(name='generator')

# 1d random noise
# Creates a 1d dense layer with 8 × 8 × 512 = 32768 values
model.add(layers.Dense(8 * 8 * 512, input_dim=LATENT_DIM))
model.add(layers.ReLU())

# Convert 1d to 3d with 512 feature channels
model.add(layers.Reshape((8, 8, 512)))

# Upsample to 16x16 using CNN
# 16×16 pixels - Bigger but shallower feature map
# 256 channels - Smaller number of channels
model.add(layers.Conv2DTranspose(256, (4, 4), strides=(2, 2), padding='same', kernel_initializer=WEIGHT_INIT))
model.add(layers.ReLU())
```

```
# Upsample to 32x32
# 32x32 pixels
# 128 channels
model.add(layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same', kernel_initializer=WEIGHT_INIT))
model.add(layers.ReLU())

# Upsample to 64x64
# 64x64 pixels
# 64 channels
model.add(layers.Conv2DTranspose(64, (4, 4), strides=(2, 2), padding='same', kernel_initializer=WEIGHT_INIT))
model.add(layers.ReLU())

# Converts (64, 64, 3) to final RGB image
model.add(layers.Conv2D(CHANNELS, (4, 4), padding='same', activation='tanh'))

generator = model
generator.summary()
```

```
Model: "generator"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 32768)             9863168

 re_lu (ReLU)                (None, 32768)             0

 reshape (Reshape)           (None, 8, 8, 512)         0

 conv2d_transpose (Conv2DTra  (None, 16, 16, 256)      2097408
 nspose)

 re_lu_1 (ReLU)              (None, 16, 16, 256)       0

 conv2d_transpose_1 (Conv2DT  (None, 32, 32, 128)      524416
 ranspose)

 re_lu_2 (ReLU)              (None, 32, 32, 128)       0

 conv2d_transpose_2 (Conv2DT  (None, 64, 64, 64)       131136
 ranspose)

 re_lu_3 (ReLU)              (None, 64, 64, 64)        0

 conv2d (Conv2D)             (None, 64, 64, 3)         3075
```

```
=================================================================
Total params: 12,619,203
Trainable params: 12,619,203
Non-trainable params: 0
_________________________________________________________________
```

### Discriminator Model

```
model = Sequential(name='discriminator')
# This controls how much negative values are kept in LeakyReLU
# This prevents "dead neurons" and helps the discriminator learn faster
alpha = 0.2

# Create conv layers
# Downsample from 64x64x3 -> 32x32x64
model.add(layers.Conv2D(64, (4, 4), strides=(2, 2), padding='same', input_shape=(64, 64, 3)))
model.add(layers.BatchNormalization())
model.add(layers.LeakyReLU(alpha=0.2))

# Downsample from 32x32x64 -> 16x16x128
# Now the network extracts higher-level patterns using 128 filters.
model.add(layers.Conv2D(128, (4, 4), strides=(2, 2), padding='same', input_shape=(64, 64, 3)))
model.add(layers.BatchNormalization())
model.add(layers.LeakyReLU(alpha=0.2))

# Downsample from 16x16x128 -> 8x8x128
# Now the network extracts higher-level patterns using 128 filters.
model.add(layers.Conv2D(128, (4, 4), strides=(2, 2), padding='same', input_shape=(64, 64, 3)))
model.add(layers.BatchNormalization())
model.add(layers.LeakyReLU(alpha=0.2))

# Downsample from 8x8x128 -> 4x4x256
model.add(layers.Conv2D(256, (4, 4), strides=(2, 2), padding='same', input_shape=(64, 64, 3)))
model.add(layers.BatchNormalization())
model.add(layers.LeakyReLU(alpha=0.2))

# Downsample from 4x4x256 -> 2x2x256
model.add(layers.Conv2D(256, (4, 4), strides=(2, 2), padding='same', input_shape=(64, 64, 3)))
model.add(layers.BatchNormalization())
model.add(layers.LeakyReLU(alpha=0.2))
```

```
# Flatten out 2x2x256 to a dense 1D array
model.add(layers.Flatten())
# Randomly sets 30% of neurons to zero during training.
# Prevents overfitting
# E.g [0.2, 0.9, 0.5, 0.7, 0.1, 0.4]
# to [0.2, 0.0, 0.5, 0.0, 0.1, 0.4]
model.add(layers.Dropout(0.3))

# Output class
# Outputs a number between 0 and 1
# E.g 0.92 -> The discriminator thinks it's a real image
# 0.03 -> The discriminator thinks it's fake
model.add(layers.Dense(1, activation='sigmoid'))

discriminator = model
discriminator.summary()
```

```
Model: "discriminator"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_1 (Conv2D)           (None, 32, 32, 64)        3136

 batch_normalization (BatchN  (None, 32, 32, 64)       256
 ormalization)

 leaky_re_lu (LeakyReLU)     (None, 32, 32, 64)        0

 conv2d_2 (Conv2D)           (None, 16, 16, 128)       131200

 batch_normalization_1 (Batc  (None, 16, 16, 128)      512
 hNormalization)

 leaky_re_lu_1 (LeakyReLU)   (None, 16, 16, 128)       0

 conv2d_3 (Conv2D)           (None, 8, 8, 128)         262272

 batch_normalization_2 (Batc  (None, 8, 8, 128)        512
 hNormalization)

 leaky_re_lu_2 (LeakyReLU)   (None, 8, 8, 128)         0
```

```
 conv2d_4 (Conv2D)           (None, 4, 4, 256)         524544

 batch_normalization_3 (Batc  (None, 4, 4, 256)        1024
 hNormalization)

 leaky_re_lu_3 (LeakyReLU)   (None, 4, 4, 256)         0

 conv2d_5 (Conv2D)           (None, 2, 2, 256)         1048832

 batch_normalization_4 (Batc  (None, 2, 2, 256)        1024
 hNormalization)

 leaky_re_lu_4 (LeakyReLU)   (None, 2, 2, 256)         0

 flatten (Flatten)           (None, 1024)              0

 dropout (Dropout)           (None, 1024)              0

 dense_1 (Dense)             (None, 1)                 1025

=================================================================
Total params: 1,974,337
Trainable params: 1,972,673
Non-trainable params: 1,664
_________________________________________________________________
```

### Create DCGAN

```
# DCGAN (Deep Convolutional Generative Adversarial Network) Wraps a generator and discriminator
# into a single Keras Model with custom training.
class DCGAN(keras.Model):
    def __init__(self, generator, discriminator, latent_dim):
        super().__init__()
        self.generator = generator
        self.discriminator = discriminator
        self.latent_dim = latent_dim
        self.g_loss_metric = keras.metrics.Mean(name='g_loss')
        self.d_loss_metric = keras.metrics.Mean(name='d_loss')

    @property
    def metrics(self):
        return [self.g_loss_metric, self.d_loss_metric]
```

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

    def train_step(self, real_images):
        # Get batch size from the data
        batch_size = tf.shape(real_images)[0]
        # Generate random noise
        random_noise = tf.random.normal(shape=(batch_size, self.latent_dim))

        # Train the discriminator with real (1) and fake (0) images
        with tf.GradientTape() as tape:
            # Compute loss on real images
            pred_real = self.discriminator(real_images, training=True)
            # Generate real image labels
            real_labels = tf.ones((batch_size, 1))
            # Label smoothing
            real_labels += 0.05 * tf.random.uniform(tf.shape(real_labels))
            d_loss_real = self.loss_fn(real_labels, pred_real)

            # Compute loss on fake images
            fake_images = self.generator(random_noise)
            pred_fake = self.discriminator(fake_images, training=True)
            # Generate fake labels
            fake_labels = tf.zeros((batch_size, 1))
            d_loss_fake = self.loss_fn(fake_labels, pred_fake)

            # Total discriminator loss
            d_loss = (d_loss_real + d_loss_fake) / 2

        # Compute discriminator gradients
        gradients = tape.gradient(d_loss, self.discriminator.trainable_variables)
        # Update the gradients
        self.d_optimizer.apply_gradients(zip(gradients, self.discriminator.trainable_variables))


        # Train the generator model
        labels = tf.ones((batch_size, 1))
        # Generator want discriminator to think that fake images are real
        with tf.GradientTape() as tape:
            # Generate fake images from generator
```

```
            fake_images = self.generator(random_noise, training=True)
            # Classify images as real or fake
            pred_fake = self.discriminator(fake_images, training=True)
            # Compute loss
            g_loss = self.loss_fn(labels, pred_fake)

        # Compute gradients
        gradients = tape.gradient(g_loss, self.generator.trainable_variables)
        # Update the gradients
        self.g_optimizer.apply_gradients(zip(gradients, self.generator.trainable_variables))

        # Update states for both models
        self.d_loss_metric.update_state(d_loss)
        self.g_loss_metric.update_state(g_loss)

        return {'d_loss': self.d_loss_metric.result(), 'g_loss': self.g_loss_metric.result()}
```

```
# DCGANMonitor is a Keras callback that periodically samples the generator
# to visualize progress and saves the generator at the end.
class DCGANMonitor(keras.callbacks.Callback):
    def __init__(self, num_imgs=25, latent_dim=300):
        self.num_imgs = num_imgs
        self.latent_dim = latent_dim
        # create random noise for generating images
        self.noise = tf.random.normal([25, latent_dim])

    def on_epoch_end(self, epoch, logs=None):
        # generate the image from noise
        g_img = self.model.generator(self.noise)
        # denormalize the image
        g_img = (g_img * 127.5) + 127.5
        g_img.numpy()

    def on_train_end(self, logs=None):
        self.model.generator.save('generator.h5')
```

## Training

```
dcgan = DCGAN(generator=generator, discriminator=discriminator, latent_dim=LATENT_DIM)
```

```
D_LR = 0.0001
G_LR = 0.0003
dcgan.compile(g_optimizer=Adam(learning_rate=G_LR, beta_1=0.5), d_optimizer=Adam(learning_rate=D_LR, beta_1=0.5), loss_fn=Bin
```

```
N_EPOCHS = 50
dcgan.fit(train_images, epochs=N_EPOCHS, callbacks=[DCGANMonitor()])
```

```
Epoch 1/50
1987/1987 [==============================] - 169s 79ms/step - d_loss: 0.4039 - g_loss: 3.5259
Epoch 2/50
1987/1987 [==============================] - 162s 82ms/step - d_loss: 0.6005 - g_loss: 1.3762
Epoch 3/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.5709 - g_loss: 1.4000
Epoch 4/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.5090 - g_loss: 1.7255
Epoch 5/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.4340 - g_loss: 2.0645
Epoch 6/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.3579 - g_loss: 2.4961
Epoch 7/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.3015 - g_loss: 2.8910
Epoch 8/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.2550 - g_loss: 3.2860
Epoch 9/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.2182 - g_loss: 3.6077
Epoch 10/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.1889 - g_loss: 3.9902
Epoch 11/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.1635 - g_loss: 4.2727
Epoch 12/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.1455 - g_loss: 4.5801
Epoch 13/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.1279 - g_loss: 4.8304
Epoch 14/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.1203 - g_loss: 5.0510
Epoch 15/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.0946 - g_loss: 5.3636
Epoch 16/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.0931 - g_loss: 5.6215
Epoch 17/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.0772 - g_loss: 5.7744
Epoch 18/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.0855 - g_loss: 5.9459
Epoch 19/50
```

```
Epoch 20/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.0621 - g_loss: 6.2972
Epoch 21/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.0587 - g_loss: 6.4810
Epoch 22/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.0494 - g_loss: 6.5973
Epoch 23/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.0386 - g_loss: 6.8496
Epoch 24/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.0284 - g_loss: 7.1225
Epoch 25/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.0238 - g_loss: 7.3169
Epoch 26/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: -0.0025 - g_loss: 7.7815
Epoch 27/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.0131 - g_loss: 7.7913
Epoch 28/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.0300 - g_loss: 7.5521
Epoch 29/50
1987/1987 [==============================] - 160s 81ms/step - d_loss: 0.0074 - g_loss: 7.7406
```

### Results

```
# Plot 49 generated images
plt.figure(figsize=(20,20))

for i in range(49):
    plt.subplot(7,7,i+1)
    # Generate random noise for each image
    noise=tf.random.normal([1,300])
    mg = dcgan.generator(noise)
    # Denormalize
    mg = (mg*127.5)+127.5

    mg.numpy()
    image = array_to_img(mg[0])

    plt.imshow(image)
    plt.axis('off')
```