from keras.layers import Input, Conv2D, MaxPooling2D, Dropout, concatenate, UpSampling2D

from keras.models import Model

from keras.optimizers import Adam

def down\_block(x, filters, kernel\_size=(3, 3), padding="same", strides=1):

c = Conv2D(filters, kernel\_size, padding=padding, strides=strides, activation="relu")(x)

c = Conv2D(filters, kernel\_size, padding=padding, strides=strides, activation="relu")(c)

p = MaxPooling2D((2, 2))(c)

p = Dropout(0.1)(p)

return c, p

def up\_block(x, skip, filters, kernel\_size=(3, 3), padding="same", strides=1):

us = UpSampling2D((2, 2))(x)

concat = concatenate([us, skip])

c = Conv2D(filters, kernel\_size, padding=padding, strides=strides, activation="relu")(concat)

c = Conv2D(filters, kernel\_size, padding=padding, strides=strides, activation="relu")(c)

c = Dropout(0.1)(c)

return c

def unet\_model(input\_size=(256, 256, 1)):

inputs = Input(input\_size)

# Down 1

c1, p1 = down\_block(inputs, 64, kernel\_size=(3, 3), padding="same", strides=1)

# Down 2

c2, p2 = down\_block(p1, 128, kernel\_size=(3, 3), padding="same", strides=1)

# Down 3

c3, p3 = down\_block(p2, 256, kernel\_size=(3, 3), padding="same", strides=1)

# Down 4

c4, p4 = down\_block(p3, 512, kernel\_size=(3, 3), padding="same", strides=1)

# Bridge

c5 = Conv2D(1024, (3, 3), padding="same")(p4)

c5 = Conv2D(1024, (3, 3), padding="same")(c5)

# Up 1

u6 = up\_block(c5, c4, 512, kernel\_size=(3, 3), padding="same", strides=1)

# Up 2

u7 = up\_block(u6, c3, 256, kernel\_size=(3, 3), padding="same", strides=1)

# Up 3

u8 = up\_block(u7, c2, 128, kernel\_size=(3, 3), padding="same", strides=1)

# Up 4

u9 = up\_block(u8, c1, 64, kernel\_size=(3, 3), padding="same", strides=1)

outputs = Conv2D(1, (1, 1), activation='sigmoid')(u9)

model = Model(inputs=[inputs], outputs=[outputs])

return model

def generator\_model():

return unet\_model()

def discriminator\_model():

inputs = Input(shape=(256, 256, 1))

x = Conv2D(64, (3, 3), padding="same")(inputs)

x = Conv2D(64, (3, 3), strides=2, padding="same")(x)

x = Dropout(0.4)(x)

x = Conv2D(128, (3, 3), padding="same")(x)

x = Conv2D(128, (3, 3), strides=2, padding="same")(x)

x = Dropout(0.4)(x)

x = Conv2D(256, (3, 3), padding="same")(x)

x = Conv2D(256, (3, 3), strides=2, padding="same")(x)

x = Dropout(0.4)(x)

x = Conv2D(512, (3, 3), padding="same")(x)

x = Conv2D(512, (3, 3), strides=2, padding="same")(x)

x = Dropout(0.4)(x)

x = Flatten()(x)

x = Dense(1, activation='sigmoid')(x)

model = Model(inputs=[inputs], outputs=[x])

return model

def gan\_model(generator, discriminator):

discriminator.trainable = False

gan\_input = Input(shape=(256, 256, 1))

generated\_image = generator(gan\_input)

gan\_output = discriminator(generated\_image)

model = Model(inputs=gan\_input, outputs=[generated\_image, gan\_output])

model.compile(loss=['binary\_crossentropy', 'binary\_crossentropy'], optimizer=Adam(lr=0.0002, beta\_1=0.5), loss\_weights=[0.5, 0.5])

return model

generator = generator\_model()

discriminator = discriminator\_model()

gan = gan\_model(generator, discriminator)

gan.summary()

In this code, we define three functions: **unet\_model** which defines the U-Net model for image segmentation, **generator\_model** which simply returns the U-Net model, and **discriminator\_model** which defines the discriminator model for the GAN. Finally, we define the **gan\_model** function which creates the complete GAN model by combining the generator and discriminator models.

In the **gan\_model** function, we first set the **trainable** property of the discriminator to **False** so that the discriminator is not trained during the training of the GAN. Then we create a new input for the GAN and pass it to the generator model to generate a new image. This generated image is then passed to the discriminator to determine whether it is a real or fake image. The final GAN model is compiled with a binary cross-entropy loss function and the Adam optimizer with a learning rate of 0.0002 and a beta value of 0.5. The loss function is weighted so that the generator and discriminator losses are balanced.