**CODING**

## MAIN.IPYNB FILE

from google.colab import drive drive.mount('/content/drive')

cd /content/drive/MyDrive/IMAGE\_NEW/

import tensorflow as tf

from tensorflow import keras import matplotlib.pyplot as plt

%matplotlib inline

from mpl\_toolkits.axes\_grid1import ImageGrid from utils.DataGenerator import Generator

from models.LGRefinementNet.LGRNet import LGRNet, dice\_coefimport os import globimport cv2

import numpy as npSIZE = 256

img\_dir = "dataset" # Enter Directory of all imagesdata\_path = os.path.join(img\_dir,'\*.jpg')

files = glob.glob(data\_path)print(files) data = []

for f1 in files:

img = cv2.imread(f1, cv2.IMREAD\_COLOR)img =

cv2.resize(img, (SIZE, SIZE)) img = cv2.cvtColor(img, cv2.COLOR\_RGB2BGR)

data.append(img) data = np.array(data)

from sklearn.model\_selection import train\_test\_split xtrain,xtest=train\_test\_split(data, test\_size=0.3) trainGenerator = Generator(xtrain, xtrain, shuffle=True)testGenerator =

Generator(xtest, xtest, shuffle=False) nSamples=19

[maskedImages, masks], originalImages = trainGenerator[nSamples] previewImage = [None]\*(len(maskedImages)+len(masks)+len(originalImages))

previewImage[::3] = originalImages previewImage[1::3] = masks previewImage[2::3] = maskedImagesfig = plt.figure(figsize=(17., 8.))

grid = ImageGrid(fig, 111, nrows\_ncols=(4, 9), axes\_pad=0.3)

for ax, image in zip(grid, previewImage):

ax.imshow(image) plt.show() keras.backend.clear\_sess ion() model = LGRNet().prepare\_mode l()model.summary()

model.compile(optimizer='adam', loss='mean\_absolute\_error', metrics=[dice\_coef])

\_ = model.fit(trainGenerator, validation\_data=testGenerator,epochs=60, steps\_per\_epoch=len(trainGenera

tor), validation\_steps=len(testGenerat or), use\_multiprocessing=True) testCases = 32

[masked\_images, masks], sample\_labels = testGenerator[54] fig, axs = plt.subplots(nrows=testCases, ncols=4, figsize=(8, 2\*testCases))for i in range(testCases): input=[masked\_images[i].reshape((1,)+masked\_images[].shape), masks[i].reshape((1,)+masks[i].shape)]

impainted\_image = model.predict(inputs) axs[i][0].imshow(masked\_ima ges[i]) axs[i][1].imshow(masks[i])

axs[i][2].imshow(impainted\_image.reshape(impainted\_image.shape[1:])) axs[i][3].imshow(sample\_labels[i])

plt.show()

## PIXEL WISE RECONSTRUCTION LOSS

import cv2

import numpy as np

img = cv2.imread('original\_image.jpg')error = np.abs(img - inpainting) error = error.sum() / (img.shape[0] \* img.shape[1] \* img.shape[2])print("Pixel-wise reconstruction error:",

error)

## CREATION OF ENCODER-DECODER ARCHITECTURE

from tensorflow import kerasimport numpy as np

from models.LGRefinementNet.RefinementConvLayer import LGNet2Dclass LGRNet: def prepare\_model(self, input\_size=(256,256,3)):input\_image = keras.layers.Input(input\_size)

input\_mask = keras.layers.Input(input\_size, name='encoder\_input')

conv1, mask1, conv2, mask2 = self. encoder\_layer(32, input\_image, input\_mask, ['conv1', 'conv2'])

conv3, mask3, conv4, mask4 = self. encoder\_layer(64, conv2, mask2, ['conv3', 'conv4']) conv5, mask5, conv6, mask6 = self. encoder\_layer(128, conv4, mask4, ['conv5', 'conv6']) conv7, mask7, conv8, mask8 = self. encoder\_layer(256, conv6, mask6, ['conv7', 'encoder\_output'])

conv9, mask9, conv10, mask10 = self. decoder\_layer(256, 128, conv8, mask8, conv7, mask7, ['conv9', 'conv10'])

conv11, mask11, conv12, mask12 = self. decoder\_layer(128, 64, conv10, mask10, conv5, mask5, ['conv11', 'conv12'])

conv13, mask13, conv14, mask14 = self. decoder\_layer(64, 32, conv12, mask12, conv3, mask3, ['conv13', 'conv14'])

conv15, mask15, conv16, mask16 = self. decoder\_layer(32, 3, conv14, mask14, conv1, mask1, ['conv15', 'decoder\_output'])

outputs = keras.layers.Conv2D(3, (3, 3), activation='sigmoid', padding='same')(conv16) return keras.models.Model(inputs=[input\_image, input\_mask], outputs=[outputs])

def encoder\_layer(self, filters, in\_layer, in\_mask, names): keras.layers.ZeroPadding2D(padding=(1,1))

conv1, mask1 = LGNet2D(64, (4,4), strides=2, padding='valid', name=names[0])([in\_layer, in\_mask])

conv1 = keras.activations.relu(conv1)

conv2, mask2 = LGNet2D(128, (4,4), strides=2, padding='valid', name=names[1])([conv1, mask1])

conv2 = keras.activations.Leakyrelu(conv2)

conv3, mask3 = LGNet2D(256, (4,4), strides=2, padding='valid',name=names[2])([conv2,mask2])

conv3 = keras.activations.Leakyrelu(conv3)

conv4, mask4 = LGNet2D(512, (4,4), strides=2, padding='valid',name=names[3])([conv3,mask3])

conv4 = keras.activations.Leakyrelu(conv4)

conv5, mask5 = LGNet2D(512, (4,4), strides=2, padding='valid',name=names[4])([conv4,mask4])

conv5 = keras.activations.Leakyrelu(conv5)conv6, mask6 = LGNet2D(512, (4,4), strides=2, padding='valid',name=names[5])([conv5,mask5])

conv6 = keras.activations.Leakyrelu(conv6)

conv7, mask7 = LGNet2D(512, (4,4), strides=2, padding='valid',name=names[6])([conv6,mask6])

conv7 = keras.activations.Leakyrelu(conv7)

conv8, mask8 = LGNet2D(512, (4,4), strides=2, padding='valid',name=names[7])([conv7,mask7])

conv8 = keras.activations.Leakyrelu(conv8)

return conv1, mask1, conv2, mask2,conv3, mask3, conv4, mask4,conv5, mask5, conv6, mask6,conv7, mask7, conv8, mask8

def decoder\_layer(self, filter1, filter2, in\_img, in\_mask, share\_img, share\_mask, names): up\_img = keras.layers.UpSampling2D(size=(2,2))(in\_img)

up\_mask = keras.layers.UpSampling2D(size=(2,2))(in\_mask) concat\_img =

keras.layers.Concatenate(axis=3)([share\_img, up\_img])

concat\_mask = keras.layers.Concatenate(axis=3)([share\_mask, up\_mask])

conv1, mask1 = LGNet2D(filter1, (4,), padding='same', name=names[0])([concat\_img, concat\_mask])

conv1 = keras.activations.relu(conv1)

conv2, mask2 = LGNet2D(filter2, (4,4), padding='same', name=names[1])([conv1, mask1])conv2 = keras.activations.relu(conv2)

conv3, mask3 = LGNet2D(filter3, (4,4), padding='same', name=names[2])([conv2,mask2])conv3= keras.activations.relu(conv3)

conv4, mask = LGNet2D(filter4, (4,4), padding='same', name=names[3])([conv3,mask3])conv4 = keras.activations.relu(conv4)

conv5, mask5 = LGNet2D(filter5, (4,4), padding='same', name=names[4])([conv4,mask4])conv5 = keras.activations.relu(conv5)

conv6, mask6 = LGNet2D(filter6, (4,4), padding='same', name=names[5])([conv5,mask5])conv6 = keras.activations.relu(conv6)

conv7, mask7 = LGNet2D(filter7, (4,4), padding='same', name=names[6])([conv6,mask6]) conv7 = keras.activations.relu(conv7)

conv8, mask8 = LGNet2D(filter8, (4,4), padding='same', name=names[7])([conv7,mask7]) conv8 = keras.activations.relu(conv8)

return conv1, mask1, conv2, mask2,conv3, mask3,conv4, mask4,conv5, mask2,conv6, mask6,conv7, mask7,conv8, mask8

## GENERATOR CLASS

from tensorflow import kerasimport numpy as np from mpl\_toolkits.axes\_grid1 import ImageGrid import matplotlib.pyplot as plt

from utils.maskImage import mask\_Imageclass Generator(keras.utils.Sequence):

def init (self, X, y, batch\_size=32, dim=(256, 256),n\_channels=3, shuffle=True):

self.batch\_size = batch\_sizeself.X = X self.y = y self.dim = dim

self.n\_channels = n\_channelsself.shuffle = shuffle self.on\_epoch\_end() def len (self):

return int(np.floor(len(self.X) / self.batch\_size)) def getitem (self, index):

indexes = self.indexes[index\*self.batch\_size:(index+1)\*self.batch\_size] X\_inputs, y\_output = self. data\_generation(indexes)

return X\_inputs, y\_output def on\_epoch\_end(self):

# Updates indexes after each epoch self.indexes = np.arange(len(self.X))if self.shuffle:

np.random.shuffle(self.indexes) def data\_generation(self, idxs):

Masked\_images = np.empty((self.batch\_size, self.dim[0], self.dim[1], self.n\_channels)) #Masked image

Mask\_batch = np.empty((self.batch\_size, self.dim[0], self.dim[1], self.n\_channels)) #

Binary Masks

y\_batch = np.empty((self.batch\_size, self.dim[0], self.dim[1], self.n\_channels)) #

Originalimage

for i, idx in enumerate(idxs): image\_copy = self.X[idx].copy() masked\_image, mask = mask\_Image(image\_copy)Masked\_images[i,] = masked\_image/255Mask\_batch[i,] = mask/255

y\_batch[i] = self.y[idx]/255

return [Masked\_images, Mask\_batch], y\_batch

## MASK IMAGE

import numpy as np import cv2

def mask\_Image(img, dim = (256,256,3)): mask = np.full(dim, 255, np.uint8) ## White backgroundfor \_ in range(np.random.randint(1, 10)):

x1, x2 = np.random.randint(1, 32), np.random.randint(1, 32)y1, y2 = np.random.randint(1, 32), np.random.randint(1, 32) thickness = np.random.randint(1, 3) cv2.line(mask,(x1,y1),(x2,y2),(0,0,0),thickness) masked\_image = img.copy() masked\_image[mask==0] = 255

return masked\_image, mask

## MASK GENERATOR

import os

from random import randint, seedimport itertools

import numpy as npimport cv2

class MaskGenerator():

def init (self, height, width, channels=3, rand\_seed=None, filepath=None):self.height = height

self.width =width self.channels=channelsself.filepath = filepath self.mask\_files = [] if self.filepath:

filenames = [f for f in os.listdir(self.filepath)]

self.mask\_files = [f for f in filenames if any(filetype in f.lower() for filetype in ['.jpeg', '.png', '.jpg'])]

print(">> Found {} masks in {}".format(len(self.mask\_files), self.filepath))# Seed for reproducibility

if rand\_seed: seed(rand\_seed) def \_generate\_mask(self):

img = np.zeros((self.height, self.width, self.channels), np.uint8)

size = int((self.width + self.height) \* 0.03)if self.width < 64 or self.height < 64: raise Exception("Width and Height of mask must be at least 64!")

# Draw random lines

for \_ in range(randint(1, 20)):

x1, x2 = randint(1, self.width), randint(1, self.width) y1, y2 = randint(1, self.height), randint(1, self.height)thickness = randint(3, size) cv2.line(img,(x1,y1),(x2,y2),(1,1,1),thickness)

# Draw random circles

for \_ in range(randint(1, 20)):

x1, y1 = randint(1, self.width), randint(1, self.height)radius = randint(3, size) cv2.circle(img,(x1,y1),radius,(1,1,1), -1)

# Draw random ellipses

for \_ in range(randint(1, 20)):

x1, y1 = randint(1, self.width), randint(1, self.height)s1, s2 = randint(1, self.width), randint(1, self.height)

a1, a2, a3 = randint(3, 180), randint(3, 180), randint(3, 180)thickness = randint(3, size) cv2.ellipse(img, (x1,y1), (s1,s2), a1, a2, a3,(1,1,1), thickness)return 1-img

def \_load\_mask(self, rotation=True, dilation=True, cropping=True): mask = cv2.imread(os.path.join(self.filepath,

np.random.choice(self.mask\_files, 1,replace=False)[0]))

if rotation:

rand = np.random.randint(-180, 180)

M = cv2.getRotationMatrix2D((mask.shape[1]/2, mask.shape[0]/2), rand, 1.5)mask = cv2.warpAffine(mask, M, (mask.shape[1], mask.shape[0]))

if dilation:

rand = np.random.randint(5, 47)

kernel = np.ones((rand, rand), np.uint8) mask = cv2.erode(mask, kernel, iterations=1)

if cropping:

x = np.random.randint(0, mask.shape[1] - self.width)y = np.random.randint(0, mask.shape[0] - self.height)

mask = mask[y:y+self.height, x:x+self.width]

return (mask > 1).astype(np.uint8) def sample(self, random\_seed=None): if random\_seed: seed(random\_seed)

if self.filepath and len(self.mask\_files) > 0:return self.\_load\_mask() else:

return self.\_generate\_mask()classImageChunker(object):

def init (self, rows, cols, overlap):self.rows = rows self.cols = cols self.overlap = overlap

def perform\_chunking(self, img\_size, chunk\_size):chunks, i = [], 0 while True:

chunks.append((i\*(chunk\_size-self.overlap/2),i\*(chunk\_size - self.overlap/2)+chunk\_size))

i+=1

if chunks[-1][1] > img\_size:break n\_count = len(chunks)

chunks[-1] = tuple(x - (n\_count\*chunk\_size - img\_size - (n\_count-1)\*self.overlap/2) forx in chunks[-1])

chunks = [(int(x), int(y)) for x, y in chunks]return chunks

def get\_chunks(self, img, scale=1):

x\_chunks, y\_chunks = [(0, self.rows)], [(0, self.cols)]if img.shape[0] > self.rows: x\_chunks = self.perform\_chunking(img.shape[0], self.rows)else:

x\_chunks = [(0, img.shape[0])]if img.shape[1] > self.cols:

y\_chunks = self.perform\_chunking(img.shape[1], self.cols)else: y\_chunks = [(0, img.shape[1])]return x\_chunks, y\_chunks

def dimension\_preprocess(self, img, padding=True):

assert len(img.shape) == 3, "Image dimension expected to be (H, W, C)"if padding:

# Check if height is too small if img.shape[0] < self.rows:

padding = np.ones((self.rows - img.shape[0], img.shape[1], img.shape[2]))img = np.concatenate((img, padding), axis=0)

# Check if width is too smallif img.shape[1] < self.cols:

padding = np.ones((img.shape[0], self.cols - img.shape[1], img.shape[2]))img = np.concatenate((img, padding), axis=1)

x\_chunks, y\_chunks = self.get\_chunks(img)

images = []

for x in x\_chunks: for y in y\_chunks:

images.append( img[x[0]:x[1], y[0]:y[1], :]

)

images = np.array(images)return images

def dimension\_postprocess(self, chunked\_images, original\_image, scale=1, padding=True): if padding:

if original\_image.shape[0] < self.rows:new\_images = []

for img in chunked\_images: new\_images.append(img[0:scale\*original\_image.shape[0], :,

:])

chunked\_images = np.array(new\_images)

if original\_image.shape[1] < self.cols:new\_images = [] for img in chunked\_images:

new\_images.append(img[:, 0:scale\*original\_image.shape[1], :])chunked\_images = np.array(new\_images)

new\_shape = ( original\_image.shape[0]\*scale,original\_image.shape[1]\*scale, original\_image.shape[2]

)

reconstruction = np.zeros(new\_shape)

x\_chunks, y\_chunks = self.get\_chunks(original\_image)i = 0 s = scale

for x in x\_chunks: for y in y\_chunks:

prior\_fill = reconstruction != 0chunk = np.zeros(new\_shape)

chunk[x[0]\*s:x[1]\*s, y[0]\*s:y[1]\*s, :] += chunked\_images[i]chunk\_fill = chunk != 0

reconstruction += chunk

reconstruction[prior\_fill & chunk\_fill] = reconstruction[prior\_fill & chunk\_fill] / 2i += 1 return reconstruction

## PARTIAL INPAINTING NETWORK

from tensorflow.keras import backend as K from tensorflow.keras.layers import InputSpec from tensorflow.keras.layers import Conv2D

class LGNet2D(Conv2D):

def init (self, \*args, n\_channels=3, mono=False, \*\*kwargs):super(). init (\*args,

\*\*kwargs)

self.input\_spec = [InputSpec(ndim=4), InputSpec(ndim=4)]

def build(self, input\_shape):

if self.data\_format == 'channels\_first': channel\_a

xis = 1else:

channel\_axis = -1

if input\_shape[0][channel\_axis] is None:

raise ValueError('The channel dimension of the inputs should be defined. Found

`None`.')

self.input\_dim = input\_shape[0][channel\_axis] kernel\_shape = self.kernel\_size + (self.input\_dim, self.filters)self.kernel = self.add\_weight(shape=kernel\_shape, initializer=self.kernel\_initializer, name='img\_kernel', regularizer=self.kernel\_regularizer, constraint=self.kernel\_constraint)

self.kernel\_mask = K.ones(shape=self.kernel\_size + (self.input\_dim, self.filters)) self.pconv\_padding = (

(int((self.kernel\_size[0]-1)/2), int((self.kernel\_size[0]-1)/2)),

(int((self.kernel\_size[0]-1)/2), int((self.kernel\_size[0]-1)/2)),

)

self.window\_size = self.kernel\_size[0] \* self.kernel\_size[1]

if self.use\_bias:

self.bias = self.add\_weight(shape=(self.filters,), initializer=self.bias\_initialize

r, name='bias', regularizer=self.bias\_regularizer, constraint=self.bias\_constraint)

else:

self.bias = Noneself.built = True

def call(self, inputs, mask=None):

if type(inputs) is not list or len(inputs) != 2:

raise Exception('RefinementConvolution2D must be called on a list of two tensors[img, mask]. Instead got: ' + str(inputs))

images = K.spatial\_2d\_padding(inputs[0], self.pconv\_padding, self.data\_format)masks = K.spatial\_2d\_padding(inputs[1], self.pconv\_padding, self.data\_format)

# Apply convolutions to maskmask\_output = K.conv2d( masks, self.kernel\_mask, strides=self.strides, padding='valid', data\_format=self.data\_format,dilation\_rate=self.dilation\_rate

)

# Apply convolutions to imageimg\_output = K.conv2d( (images\*masks), self.kernel, strides=self.strides, padding='valid', data\_format=self.data\_format,dilation\_rate=self.dilation\_rate

)

mask\_ratio = self.window\_size / (mask\_output + 1e-8)mask\_output = K.clip(mask\_output, 0, 1)

mask\_ratio = mask\_ratio \* mask\_outputimg\_output = img\_output \* mask\_ratio if self.use\_bias:

img\_output = K.bias\_add(img\_output, self.bias, data\_format=self.data\_format)

if self.activation is not None:

img\_output = self.activation(img\_output)return [img\_output, mask\_output]

def compute\_output\_shape(self, input\_shape):if self.data\_format == 'channels\_last': space = input\_shape[0][1:-1]new\_space = []

for i in range(len(space)): new\_dim = conv\_output\_length( space[i], self.kernel\_size[i]padding='same', stride=self.strides[i], dilation=self.dilation\_rate[i])

new\_space.append(new\_dim)

new\_shape = (input\_shape[0][0],) + tuple(new\_space) + (self.filters,)return [new\_shape, new\_shape]

if self.data\_format == 'channels\_first':space = input\_shape[2:] new\_space = []

for i in range(len(space)): new\_dim = conv\_output\_length( space[i], self.kernel\_size[i], padding='same', stride=self.strides[i], dilation=self.dilation\_rate[i])new\_space.append(new\_dim)

new\_shape = (input\_shape[0], self.filters) + tuple(new\_space)return [new\_shape, new\_shape]

def conv\_output\_length(input\_length, filter\_size,padding, stride, dilation=1): if input\_length is None:return None

assert padding in {'same', 'valid', 'full', 'causal'} dilated\_filter\_size = (filter\_size - 1) \*

dilation + 1if padding == 'same':

output\_length = input\_lengthelif padding == 'valid':

output\_length = input\_length - dilated\_filter\_size + 1elif padding == 'causal': output\_length = input\_lengthelif padding == 'full':

output\_length = input\_length + dilated\_filter\_size - 1return (output\_length + stride - 1) // stride

## DISCRIMINATOR

import tensorflow as tf

class SpectralNormalization(tf.keras.layers.Wrapper): def init (self, layer, iteration=1, eps=1e-12,

\*\*kwargs): super(SpectralNormalization, self). init (layer, \*\*kwargs)self.iteration = iteration

self.eps = eps

def build(self, input\_shape): self.layer.build(input\_shape self.w = self.layer.kernel self.w\_shape = self.w.shape.as\_list()

self.u=self.add\_weight(shape=(1,self.w\_shape[-1]), initializer=tf.initializers.TruncatedNormal(), trainable=False, name='sn\_u', dtype=tf.float32)

super(SpectralNormalization, self).build()

def call(self, inputs):

for i in range(self.iteration):

w\_reshaped = tf.reshape(self.w, [-1, self.w\_shape[- 1]])eps = tf.constant(self.eps, dtype=tf.float32)

v = tf.linalg.l2\_normalize(tf.matmul(self.u, tf.transpose(w\_reshaped)))u = tf.linalg.l2\_normalize(tf.matmul(v, w\_reshaped))

self.u.assign(u)

sigma = tf.matmul(tf.matmul(v, w\_reshaped), tf.transpose(u)) self.layer.kernel = self.w / sigma

return self.layer(inputs)

def make\_discriminator\_model(input\_shape):model = tf.keras.Sequential() model.add(SpectralNormalization(tf.keras.layers.Conv2D(64,(5,5),strides=(2,2),

padding='same'), input\_shape=input\_shape)) model.add(tf.keras.layers.LeakyReLU(alpha=0.2)) model.add(SpectralNormalization(tf.keras.layers.Conv2D(128,(5,5),strides=(2, 2), padding='same'))) model.add(tf.keras.layers.LeakyReLU(alpha=0.2)) model.add(tf.keras.layers.Flatten()) model.add(SpectralNormalization(tf.keras.layers.Dense(1)))

return model

## SMALL REFINEMENT NETWORK

from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D,

Concatenate, Dropout input\_shape = (256,

256, 3)

input\_img = Input(shape=input\_shape)

conv1 = Conv2D(32, (3, 3), activation='relu',

padding='same')(input\_img)conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(conv1) pool1 = MaxPooling2D(pool\_size=(2, 2))(conv1)

conv2 = Conv2D(64, (3, 3), activation='relu',

padding='same')(pool1) conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv2) pool2 = MaxPooling2D(pool\_size=(2, 2))(conv2)

conv3 = Conv2D(128, (3, 3), activation='relu',

padding='same')(pool2) conv3 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv3) pool3 = MaxPooling2D(pool\_size=(2, 2))(conv3)

conv4 = Conv2D(256, (3, 3), activation='relu', padding='same')(pool3)

conv4 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv4) drop4 = Dropout(0.5)(conv4)

pool4 = MaxPooling2D(pool\_size=(2, 2))(drop4)

conv5 = Conv2D(512, (3, 3), activation='relu', padding='same')(pool4) conv5 = Conv2D(512, (3, 3), activation='relu', padding='same')(conv5)drop5 = Dropout(0.5)(conv5)

up6 = Conv2D(256, (2, 2), activation='relu', padding='same')(UpSampling2D(size=(2, 2))(drop5))

merge6 = Concatenate(axis=3)([drop4, up6])

conv6 = Conv2D(256, (3, 3), activation='relu', padding='same')(merge6)conv6 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv6)

up7 = Conv2D(128, (2, 2), activation='relu', padding='same')(UpSampling2D(size=(2, 2))(conv6))

merge7 = Concatenate(axis=3)([conv3, up7])

conv7 = Conv2D(128, (3, 3), activation='relu', padding='same')(merge7)conv7 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv7) up8=Conv2D(64,(2, 2),activation='relu',padding='same')(UpSampling2D(size=(2, 2))(conv7))

merge8 = Concatenate(axis=3)([conv2, up8])

conv8 = Conv2D(64, (3, 3), activation='relu', padding='same')(merge8)conv8 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv8)up9 =Conv2D(32, (2,2),

activation='relu', padding='same')(UpSampling2D(size=(2,2))(conv8)) merge9 = Concatenate(axis=3)([conv1, up9])conv9 = Conv2D(lrfn = Conv2D(32, (3, 3), activation='relu', padding='same')(conv9)lrfn = Conv2D(32, (3, 3), activation='relu', padding='same')(lrfn) lrfn = Conv2D(32, (3, 3), activation='relu', padding='same')(lrfn)

output\_layer = Conv2D(num\_classes, (1, 1), activation='softmax')(lrfn)model =

Model(inputs=input\_img, outputs=output\_layer) model.compile(optimizer=Adam(lr=1e-4),loss='categorical\_crossentropy', metrics=['accuracy'])

## LOSS FUNCTIONS

import tensorflow as tfimport numpy as np import PIL.Image import matplotlib.pyplot as plt

vgg = tf.keras.applications.VGG19(include\_top=False, weights='imagenet')vgg.trainable = False

style\_layers = ['block1\_conv1', 'block2\_conv1', 'block3\_conv1', 'block4\_conv1','block5\_conv1']

content\_layers = ['block5\_conv2'] num\_style\_layers = len(style\_layers)def load\_image(image\_path):

img = tf.keras.preprocessing.image.load\_img(image\_path)img = tf.keras.preprocessing.image.img\_to\_array(img) img = tf.keras.applications.vgg19.preprocess\_input(img) img = np.expand\_dims(img, axis=0) return img

def show\_image(img):

img = np.clip(img, 0, 255).astype('uint8')plt.imshow(img) plt.axis('off')plt.show()

# Compute the gram matrix for a feature mapdef gram\_matrix(input\_tensor): result = tf.linalg.einsum('bijc,bijd->bcd', input\_tensor, input\_tensor)input\_shape = tf.shape(input\_tensor)

num\_locations = tf.cast(input\_shape[1]\*input\_shape[2], tf.float32)return result/(num\_locations)

# Compute the style loss for a single layerdef style\_loss(style, combination): style\_gram = gram\_matrix(style) combination\_gram = gram\_matrix(combination)

style\_loss = tf.reduce\_mean(tf.square(style\_gram - combination\_gram))return style\_loss

def perceptual\_loss(content, combination):

content\_loss = tf.reduce\_mean(tf.square(content - combination))return content\_loss

style\_image = load\_image('image1.jpg') content\_image = load\_image('image2.jpg')import tensorflow as tf

from tensorflow import keras

from tensorflow.keras.preprocessing.image import load\_img, img\_to\_arrayfrom tensorflow.keras.utils import normalize

# Load the input image and mask image input\_img = load\_img('input\_image.jpg') mask\_img = load\_img('mask\_image.jpg')

input\_array = normalize(img\_to\_array(input\_img))mask\_array = normalize(img\_to\_array(mask\_img))

model = keras.models.load\_model('image\_inpainting\_model.h5')predicted\_image = model.predict([input\_array, mask\_array])

ground\_truth\_img = load\_img('ground\_truth\_image.jpg') ground\_truth\_array = normalize(img\_to\_array(ground\_truth\_img))

mse = tf.keras.losses.mean\_squared\_error(ground\_truth\_array, predicted\_image)

## LARGE REFINEMENT NETWORK

import tensorflow as tf

from tensorflow.keras.layers import Input, Conv2D, BatchNormalization, Activation, Add,

UpSampling2D, Concatenate

from tensorflow.keras.models import Model

def conv\_block(inputs, filters, kernel\_size):

x = tf.keras.layers.Conv2D(filters, kernel\_size, padding='same')(inputs)x = tf.keras.layers.BatchNormalization()(x)

x = tf.keras.layers.ReL U()(x)return x

def grn\_attention(inputs, n\_filters):

global\_features = tf.reduce\_mean(inputs, axis=[1, 2], keepdims=True)global\_features = conv\_block(global\_features, n\_filters, 1) local\_features = conv\_block(inputs, n\_filters, 3)

att = tf.keras.layers.Conv2D(n\_filters, 1, activation='sigmoid')(local\_features) att\_features = tf.multiply(local\_features, att)

refinement = tf.concat([global\_features, att\_features], axis=-1)refinement = conv\_block(refinement, n\_filters, 1)

output = tf.add(inputs, refinement)return output

def grn\_with\_attention(input\_shape, n\_filters, n\_blocks):inputs = tf.keras.layers.Input(shape=input\_shape)

x = conv\_block(inputs, n\_filters, 3)for i in range(n\_blocks):

x = grn\_attention(x, n\_filters)

outputs = conv\_block(x, input\_shape[-1], 3) model = tf.keras.Model(inputs=inputs, outputs=outputs)return model

input\_shape = (256, 256, 3)

n\_filters = 64

n\_blocks = 8

model = grn\_with\_attention(input\_shape, n\_filters, n\_blocks)

model.summary()