**OB v.2 Algorithm Codes**

**Main.py**

import tensorflow as tf

import matplotlib.pyplot as plt

import os

from tensorflow.keras import layers

import time

import glob

import numpy as np

import cv2

from IPython import display

IN\_WIDTH = int(320)

IN\_HEIGHT = int(240)

BUFFER\_SIZE = 950

BATCH\_SIZE = 5

EPOCHS = 75

FOLDER = './crop{}\_main/\*'.format(str(IN\_HEIGHT))

#input\_list = ['1','2','3','4','5','6','7','8','9','10']

noise\_dim = 1000

num\_examples\_to\_generate = 16

IMAGE\_COUNT = 95 # len(input\_list)

train\_images = np.zeros((IMAGE\_COUNT, IN\_HEIGHT, IN\_WIDTH, 1))

imList = glob.glob(FOLDER)

for i, file in enumerate(imList):

str = os.path.split(file)

index = str[-1][7:-4] # Take file number

#if index in ['1','2','3','4','5','6','7','8','9','10']:

# train\_images[int(index)-1, :, :, 0] = cv2.imread(file, cv2.IMREAD\_GRAYSCALE)

train\_images[i, :, :, 0] = cv2.imread(file, cv2.IMREAD\_GRAYSCALE)

train\_images = np.repeat(train\_images, 10, axis=0)

train\_images = train\_images.reshape((BUFFER\_SIZE, IN\_HEIGHT, IN\_WIDTH, 1)).astype('float32')

train\_images = (train\_images - 127.5) / 127.5 # Normalize the images to [-1, 1]

# Batch and shuffle the data

train\_dataset = tf.data.Dataset.from\_tensor\_slices(train\_images).shuffle(BUFFER\_SIZE).batch(BATCH\_SIZE)

"""for image\_batch in train\_dataset:

print(image\_batch.shape)

#plt.imshow(image\_batch[0,:,:,0], cmap='gray')

plt.show()

"""

def make\_generator\_model():

model = tf.keras.Sequential()

model.add(layers.Dense(IN\_HEIGHT/4\*IN\_WIDTH/4\*64, use\_bias=False, input\_shape=(noise\_dim,)))

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Reshape((int(IN\_HEIGHT/4), int(IN\_WIDTH/4), 64)))

assert model.output\_shape == (None, int(IN\_HEIGHT/4), int(IN\_WIDTH/4), 64) # Note: None is the batch size

model.add(layers.Conv2DTranspose(32, (5, 5), strides=(1, 1), padding='same', use\_bias=False))

assert model.output\_shape == (None, int(IN\_HEIGHT/4), int(IN\_WIDTH/4), 32)

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(16, (5, 5), strides=(2, 2), padding='same', use\_bias=False))

assert model.output\_shape == (None, int(IN\_HEIGHT/2), int(IN\_WIDTH/2), 16)

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use\_bias=False, activation='tanh'))

assert model.output\_shape == (None, IN\_HEIGHT, IN\_WIDTH, 1)

return model

def make\_discriminator\_model():

model = tf.keras.Sequential()

model.add(layers.Conv2D(16, (5, 5), strides=(2, 2), padding='same',

input\_shape=[IN\_HEIGHT, IN\_WIDTH, 1]))

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.1))

model.add(layers.Conv2D(32, (5, 5), strides=(2, 2), padding='same'))

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.1))

model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same'))

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.1))

model.add(layers.Flatten())

model.add(layers.Dense(1))

return model

generator = make\_generator\_model()

discriminator = make\_discriminator\_model()

# This method returns a helper function to compute cross entropy loss

cross\_entropy = tf.keras.losses.BinaryCrossentropy(from\_logits=True)

def discriminator\_loss(real\_output, fake\_output):

real\_loss = cross\_entropy(tf.ones\_like(real\_output), real\_output)

fake\_loss = cross\_entropy(tf.zeros\_like(fake\_output), fake\_output)

total\_loss = real\_loss + fake\_loss

return total\_loss

def generator\_loss(fake\_output):

return cross\_entropy(tf.ones\_like(fake\_output), fake\_output)

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=generator\_optimizer,

discriminator\_optimizer=discriminator\_optimizer,

generator=generator,

discriminator=discriminator)

# We will reuse this seed overtime (so it's easier)

# to visualize progress in the animated GIF)

seed = tf.random.normal([num\_examples\_to\_generate, noise\_dim])

# Notice the use of `tf.function`

# This annotation causes the function to be "compiled".

@tf.function

def train\_step(images):

noise = tf.random.normal([BATCH\_SIZE, noise\_dim])

with tf.GradientTape() as gen\_tape, tf.GradientTape() as disc\_tape:

generated\_images = generator(noise, training=True)

real\_output = discriminator(images, training=True)

fake\_output = discriminator(generated\_images, training=True)

gen\_loss = generator\_loss(fake\_output)

disc\_loss = discriminator\_loss(real\_output, fake\_output)

gradients\_of\_generator = gen\_tape.gradient(gen\_loss, generator.trainable\_variables)

gradients\_of\_discriminator = disc\_tape.gradient(disc\_loss, discriminator.trainable\_variables)

generator\_optimizer.apply\_gradients(zip(gradients\_of\_generator, generator.trainable\_variables))

discriminator\_optimizer.apply\_gradients(zip(gradients\_of\_discriminator, discriminator.trainable\_variables))

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,

seed)

"""

# Save the model every 20 epochs

if (epoch + 1) % 20 == 0:

checkpoint.save(file\_prefix = checkpoint\_prefix)"""

print('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

print('Last gen loss: {} and disc loss: {}'.format(gen\_loss, disc\_loss))

def generate\_and\_save\_images(model, epoch, test\_input):

# Notice `training` is set to False.

# This is so all layers run in inference mode (batchnorm).

predictions = model(test\_input, training=False)

fig = plt.figure(figsize=(4,4))

for i in range(predictions.shape[0]):

plt.subplot(4, 4, i+1)

plt.imshow(predictions[i, :, :, 0] \* 127.5 + 127.5, cmap='gray')

plt.axis('off')

img = predictions[i, :, :, 0] \* 127.5 + 127.5

if epoch == EPOCHS:

cv2.imwrite("./result/res{}.png".format(i+1), img.numpy())

plt.savefig('./result/image\_at\_epoch\_{:04d}.png'.format(epoch))

plt.close()

# MAIN FUNCTION

#generator.summary()

train(train\_dataset, EPOCHS)

checkpoint.save(file\_prefix = checkpoint\_prefix)

**Main app.py**

import tensorflow as tf

import matplotlib.pyplot as plt

import os

from tensorflow.keras import layers

import time

import glob

import numpy as np

import cv2

IN\_WIDTH = int(320)

IN\_HEIGHT = int(240)

BUFFER\_SIZE = 1000

BATCH\_SIZE = 10

EPOCHS = 50

LEARN\_RATE = 1e-4

CONV\_SIZE = (3, 3)

START\_SHAPE = 61

RANGE\_SHAPE = 2

NEG\_IMAGES = 5

FOLDER = './crop{}/\*'.format(str(IN\_HEIGHT))

input\_list = [ 1, 8, 18, 29, 44, 59, 71, 84, 99, 113,

121, 133, 144, 154, 164, 175, 186, 198, 211, 224,

235, 245, 256, 262, 272, 281, 293, 301, 309, 319, 329,

337, 345, 347, 361, 371, 386, 396, 411, 421, 430,

439, 450, 458, 477, 485, 502, 517, 524, 535, 554,

574, 592, 601, 616, 631, 641, 652, 666, 679, 693,

703, 713, 728, 738, 751, 767, 782, 796, 811, 821,

832, 841, 849, 861, 871, 880, 891, 901, 912, 922,

933, 944, 957, 966, 980, 991, 1007, 1021, 1031, 1043,

1053, 1061, 1071, 1081, 1091, 1101, 1111, 1121, 1131]

noise\_dim = 1000

num\_examples\_to\_generate = 20

def make\_generator\_model():

model = tf.keras.Sequential()

model.add(layers.Dense(IN\_HEIGHT\*IN\_WIDTH\*4, use\_bias=False, input\_shape=(noise\_dim,)))

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Reshape((int(IN\_HEIGHT/4), int(IN\_WIDTH/4), 64)))

assert model.output\_shape == (None, int(IN\_HEIGHT/4), int(IN\_WIDTH/4), 64) # Note: None is the batch size

model.add(layers.Conv2DTranspose(32, CONV\_SIZE, strides=(1, 1), padding='same', use\_bias=False))

assert model.output\_shape == (None, int(IN\_HEIGHT/4), int(IN\_WIDTH/4), 32)

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(16, CONV\_SIZE, strides=(2, 2), padding='same', use\_bias=False))

assert model.output\_shape == (None, int(IN\_HEIGHT/2), int(IN\_WIDTH/2), 16)

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(1, CONV\_SIZE, strides=(2, 2), padding='same', use\_bias=False, activation='tanh'))

assert model.output\_shape == (None, IN\_HEIGHT, IN\_WIDTH, 1)

return model

def make\_discriminator\_model():

model = tf.keras.Sequential()

model.add(layers.Conv2D(16, CONV\_SIZE, strides=(1, 1), padding='same',

input\_shape=[IN\_HEIGHT, IN\_WIDTH, 1]))

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.3))

model.add(layers.Conv2D(32, CONV\_SIZE, strides=(2, 2), padding='same'))

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.3))

model.add(layers.Conv2D(64, CONV\_SIZE, strides=(2, 2), padding='same'))

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.3))

model.add(layers.Flatten())

model.add(layers.Dense(1))

return model

generator = make\_generator\_model()

generator.save\_weights('gen.h5')

discriminator = make\_discriminator\_model()

discriminator.save\_weights('disc.h5')

cross\_entropy = tf.keras.losses.BinaryCrossentropy(from\_logits=True)

def discriminator\_loss(real\_output, fake\_output):

real\_loss = cross\_entropy(tf.ones\_like(real\_output), real\_output)

fake\_loss = cross\_entropy(tf.zeros\_like(fake\_output), fake\_output)

total\_loss = real\_loss + fake\_loss

return total\_loss

def generator\_loss(fake\_output):

return cross\_entropy(tf.ones\_like(fake\_output), fake\_output)

generator\_optimizer = tf.keras.optimizers.Adam(LEARN\_RATE)

discriminator\_optimizer = tf.keras.optimizers.Adam(LEARN\_RATE)

@tf.function

def train\_step(images):

noise = tf.random.normal([BATCH\_SIZE, noise\_dim])

with tf.GradientTape() as gen\_tape, tf.GradientTape() as disc\_tape:

generated\_images = generator(noise, training=True)

real\_output = discriminator(images, training=True)

fake\_output = discriminator(generated\_images, training=True)

gen\_loss = generator\_loss(fake\_output)

disc\_loss = discriminator\_loss(real\_output, fake\_output)

gradients\_of\_generator = gen\_tape.gradient(gen\_loss, generator.trainable\_variables)

gradients\_of\_discriminator = disc\_tape.gradient(disc\_loss, discriminator.trainable\_variables)

generator\_optimizer.apply\_gradients(zip(gradients\_of\_generator, generator.trainable\_variables))

discriminator\_optimizer.apply\_gradients(zip(gradients\_of\_discriminator, discriminator.trainable\_variables))

return gen\_loss, disc\_loss

def train(dataset, epochs, shape\_no):

test\_input = tf.random.normal([4, noise\_dim])

for epoch in range(epochs):

start = time.time()

for image\_batch in dataset:

gen\_loss, disc\_loss = train\_step(image\_batch)

if (epoch + 1) % 5 == 0:

predictions = generator(test\_input, training=False)

fig = plt.figure(figsize=(4, 4))

for i in range(predictions.shape[0]):

plt.subplot(2, 2, i + 1)

plt.imshow(predictions[i, :, :, 0] \* 127.5 + 127.5, cmap='gray')

plt.axis('off')

if i == 0:

plt.title("Gen Loss: {:0.3f} Disc Loss: {:0.3f}".format(gen\_loss, disc\_loss), loc='left')

plt.savefig('./result/image\_{}\_epoch\_{:04d}.png'.format(shape\_no, epoch+1))

plt.close()

print('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

print('Last gen loss: {} and disc loss: {}'.format(gen\_loss, disc\_loss))

def generate\_and\_save\_images(model, test\_input, shape\_no):

predictions = model(test\_input, training=False)

for i in range(predictions.shape[0]):

img = predictions[i, :, :, 0] \* 127.5 + 127.5

cv2.imwrite("./result/res{}\_{}.png".format(shape\_no, i+1), img.numpy())

# MAIN FUNCTION #

for m, n in enumerate(input\_list):

if m >= START\_SHAPE:

IMAGE\_COUNT = input\_list[m+1] - n + NEG\_IMAGES

train\_images = np.zeros((IMAGE\_COUNT, IN\_HEIGHT, IN\_WIDTH, 1))

imList = glob.glob(FOLDER)

for file in imList:

str = os.path.split(file)

index = str[-1][7:-4] # Take file number

if int(index) in range(n, input\_list[m+1]):

train\_images[int(index)-n, :, :, 0] = cv2.imread(file, cv2.IMREAD\_GRAYSCALE)

negList = glob.glob("./neg/\*.png")

negList = np.random.choice(negList, 1, False)

"""for dummy, file in enumerate(negList):

train\_images[IMAGE\_COUNT-NEG\_IMAGES+dummy, :, :, 0] = cv2.imread(file, cv2.IMREAD\_GRAYSCALE)"""

for dummy in range(NEG\_IMAGES):

print(negList[0])

train\_images[IMAGE\_COUNT - NEG\_IMAGES + dummy, :, :, 0] = cv2.imread(negList[0], cv2.IMREAD\_GRAYSCALE)

train\_images = np.tile(train\_images, (np.ceil(BUFFER\_SIZE / IMAGE\_COUNT).astype('int'), 1, 1, 1))

train\_images = train\_images.reshape((train\_images.shape[0], IN\_HEIGHT, IN\_WIDTH, 1)).astype('float32')

train\_images = (train\_images - 127.5) / 127.5 # Normalize the images to [-1, 1]

train\_dataset = tf.data.Dataset.from\_tensor\_slices(train\_images[0:BUFFER\_SIZE]).shuffle(BUFFER\_SIZE).batch(BATCH\_SIZE)

time\_train = time.time()

train(train\_dataset, EPOCHS, m+1)

print("Total Training Time: {:0.3f} min \n".format((time.time()-time\_train) / 60.0))

seed = tf.random.normal([num\_examples\_to\_generate, noise\_dim])

generate\_and\_save\_images(generator, seed, m+1)

generator.load\_weights("gen.h5")

discriminator.load\_weights("disc.h5")

if m+1 == START\_SHAPE+RANGE\_SHAPE:

break

**Generate Result**

import cv2

import tensorflow as tf

from tensorflow.keras import layers

IN\_WIDTH = int(320)

IN\_HEIGHT = int(240)

noise\_dim = 1000

def make\_generator\_model():

model = tf.keras.Sequential()

model.add(layers.Dense(IN\_HEIGHT/4\*IN\_WIDTH/4\*64, use\_bias=False, input\_shape=(noise\_dim,)))

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Reshape((int(IN\_HEIGHT/4), int(IN\_WIDTH/4), 64)))

assert model.output\_shape == (None, int(IN\_HEIGHT/4), int(IN\_WIDTH/4), 64) # Note: None is the batch size

model.add(layers.Conv2DTranspose(32, (5, 5), strides=(1, 1), padding='same', use\_bias=False))

assert model.output\_shape == (None, int(IN\_HEIGHT/4), int(IN\_WIDTH/4), 32)

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(16, (5, 5), strides=(2, 2), padding='same', use\_bias=False))

assert model.output\_shape == (None, int(IN\_HEIGHT/2), int(IN\_WIDTH/2), 16)

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use\_bias=False, activation='tanh'))

assert model.output\_shape == (None, IN\_HEIGHT, IN\_WIDTH, 1)

return model

def make\_discriminator\_model():

model = tf.keras.Sequential()

model.add(layers.Conv2D(16, (5, 5), strides=(2, 2), padding='same',

input\_shape=[IN\_HEIGHT, IN\_WIDTH, 1]))

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.1))

model.add(layers.Conv2D(32, (5, 5), strides=(2, 2), padding='same'))

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.1))

model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same'))

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.1))

model.add(layers.Flatten())

model.add(layers.Dense(1))

return model

generator = make\_generator\_model()

discriminator = make\_discriminator\_model()

generator\_optimizer = tf.keras.optimizers.Adam(1e-4)

discriminator\_optimizer = tf.keras.optimizers.Adam(1e-4)

checkpoint\_dir = './training\_checkpoints'

checkpoint = tf.train.Checkpoint(generator\_optimizer=generator\_optimizer,

discriminator\_optimizer=discriminator\_optimizer,

generator=generator,

discriminator=discriminator)

checkpoint.restore(tf.train.latest\_checkpoint(checkpoint\_dir))

test\_input = tf.random.normal([50, noise\_dim])

predictions = generator(test\_input, training=False)

for i in range(predictions.shape[0]):

img = predictions[i, :, :, 0] \* 127.5 + 127.5

cv2.imwrite("./result/res{}.png".format(i+1), img.numpy())

**Draw Brush**

import cv2

import glob

import matplotlib.pyplot as plt

import numpy as np

import random

def show\_img(img):

plt.imshow(img)

plt.show()

def find\_alpha\_comp (alpha, beta):

return alpha + np.multiply(beta, 1-alpha).astype('float32')

def find\_color\_comp (ca, cb, aa, ab):

trans\_mat = find\_alpha\_comp(aa, ab)

nom = np.multiply(ca, aa) + np.multiply(np.multiply(cb, ab), (1 - aa))

return np.divide(nom, trans\_mat, out=np.zeros\_like(nom), where=trans\_mat!=0)

folder = 'FINAL\_OB2'

folder\_in = 'INPUT\_MAIN'

folder\_neg = 'FINAL\_NEG'

folder\_res = 'FINAL\_RESULT'

imList = glob.glob('./{}/res\*'.format(folder))

kernel = np.ones((7, 7), dtype='float32')

cl = cv2.createCLAHE(2,(8, 8))

for out\_no in range(0, 10):

shape\_list = list(range(1, 21))+list(range(26, 32))+list(range(37, 41))+[62, 63, 70, 71]

shape\_list = np.sort((np.random.choice(shape\_list, 10, False)))

file\_list = []

for shape in shape\_list:

idx = random.choice(range(1, 21))

file\_list.append('{}\_{}'.format(shape, idx))

white = np.ones((3508, 4961, 4), dtype='float32')

white[:, :, 3] = 0

white\_in = np.ones((3508, 4961, 4), dtype='float32')

white\_in[:, :, 3] = 0

white\_neg = np.ones((3508, 4961, 4), dtype='float32')

white\_neg[:, :, 3] = 0

for ind, file in enumerate(file\_list):

IMG\_SIZE = np.array([1, 1]) \* np.random.randint(750, 2000)

all\_255 = np.ones(IMG\_SIZE, dtype='float32')

pt = (np.random.randint(0, 4960 - IMG\_SIZE[1]), np.random.randint(0, 3507 - IMG\_SIZE[0]))

raw\_in = cv2.imread("./{}/input{}.png".format(folder\_in, shape\_list[ind]))

raw\_neg = cv2.imread("./{}/res{}.png".format(folder\_neg, file))

raw = cv2.imread("./{}/res{}.png".format(folder, file))

img\_in = cv2.cvtColor(raw\_in, cv2.COLOR\_BGR2GRAY)

img\_in = cv2.resize(img\_in, (IMG\_SIZE[0], IMG\_SIZE[1]), cv2.INTER\_CUBIC)

img\_in = (img\_in / 255.0)

img\_neg = cv2.cvtColor(raw\_neg, cv2.COLOR\_BGR2GRAY)

img\_neg = cv2.resize(img\_neg, (IMG\_SIZE[0], IMG\_SIZE[1]), cv2.INTER\_CUBIC)

img\_neg = (img\_neg / 255.0)

img = cv2.cvtColor(raw, cv2.COLOR\_BGR2GRAY)

img = cl.apply(img)

img = cv2.resize(img, (IMG\_SIZE[0], IMG\_SIZE[1]), cv2.INTER\_CUBIC)

img = cv2.morphologyEx(img, cv2.MORPH\_OPEN, kernel)

img = cv2.morphologyEx(img, cv2.MORPH\_CLOSE, kernel)

img = (img/255.0)

bright = (np.random.rand() + 1.5) # Random saturation

neg\_in = (1 - img\_in)

neg\_in = np.minimum(neg\_in, all\_255).astype('float32')

neg\_neg = (1 - img\_neg)\*bright

neg\_neg = np.minimum(neg\_neg, all\_255).astype('float32')

neg = (1 - img)\*bright

neg = np.minimum(neg, all\_255).astype('float32')

mask\_in = np.array(neg\_in)

mask\_in[mask\_in != 0] = 1

clipped\_in = white\_in[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], :]

mask\_neg = np.array(neg\_neg)

mask\_neg[mask\_neg != 0] = 1

clipped\_neg = white\_neg[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], :]

mask = np.array(neg)

mask[mask != 0] = 1

clipped = white[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], :]

chroma = np.random.rand(3)

ch\_r\_in = find\_color\_comp(mask\_in \* chroma[0], clipped\_in[:, :, 2], neg\_in, clipped\_in[:, :, 3])

ch\_g\_in = find\_color\_comp(mask\_in \* chroma[1], clipped\_in[:, :, 1], neg\_in, clipped\_in[:, :, 3])

ch\_b\_in = find\_color\_comp(mask\_in \* chroma[2], clipped\_in[:, :, 0], neg\_in, clipped\_in[:, :, 3])

ch\_a\_in = find\_alpha\_comp(neg\_in, clipped\_in[:, :, 3])

ch\_r\_neg = find\_color\_comp(mask\_neg \* chroma[0], clipped\_neg[:, :, 2], neg\_neg, clipped\_neg[:, :, 3])

ch\_g\_neg = find\_color\_comp(mask\_neg \* chroma[1], clipped\_neg[:, :, 1], neg\_neg, clipped\_neg[:, :, 3])

ch\_b\_neg = find\_color\_comp(mask\_neg \* chroma[2], clipped\_neg[:, :, 0], neg\_neg, clipped\_neg[:, :, 3])

ch\_a\_neg = find\_alpha\_comp(neg\_neg, clipped\_neg[:, :, 3])

ch\_r = find\_color\_comp(mask\* chroma[0], clipped[:, :, 2], neg, clipped[:, :, 3])

ch\_g = find\_color\_comp(mask\* chroma[1], clipped[:, :, 1], neg, clipped[:, :, 3])

ch\_b = find\_color\_comp(mask\* chroma[2], clipped[:, :, 0], neg, clipped[:, :, 3])

ch\_a = find\_alpha\_comp(neg, clipped[:, :, 3])

white\_in[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 0] = ch\_b\_in

white\_in[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 1] = ch\_g\_in

white\_in[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 2] = ch\_r\_in

white\_in[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 3] = ch\_a\_in

white\_neg[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 0] = ch\_b\_neg

white\_neg[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 1] = ch\_g\_neg

white\_neg[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 2] = ch\_r\_neg

white\_neg[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 3] = ch\_a\_neg

white[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 0] = ch\_b

white[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 1] = ch\_g

white[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 2] = ch\_r

white[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 3] = ch\_a

white = 255 \* white

white\_in = 255 \* white\_in

white\_neg = 255 \* white\_neg

cv2.imwrite("./{}/final\_out{}.png".format(folder\_res, out\_no+1), white)

cv2.imwrite("./{}/final\_neg{}.png".format(folder\_res, out\_no + 1), white\_neg)

cv2.imwrite("./{}/final\_in{}.png".format(folder\_res, out\_no+1), white\_in)

**Convert Image**

import cv2

import glob

import matplotlib.pyplot as plt

import os

import numpy as np

def read\_transparent\_png(filename):

image\_4channel = cv2.imread(filename, cv2.IMREAD\_UNCHANGED)

alpha\_channel = image\_4channel[:,:,3]

rgb\_channels = image\_4channel[:,:,:3]

# White Background Image

white\_background\_image = np.ones\_like(rgb\_channels, dtype=np.uint8) \* 255

# Alpha factor

alpha\_factor = alpha\_channel[:,:,np.newaxis].astype(np.float32) / 255.0

alpha\_factor = np.concatenate((alpha\_factor,alpha\_factor,alpha\_factor), axis=2)

# Transparent Image Rendered on White Background

base = rgb\_channels.astype(np.float32) \* alpha\_factor

white = white\_background\_image.astype(np.float32) \* (1 - alpha\_factor)

final\_image = base + white

final\_image = cv2.cvtColor(final\_image, cv2.COLOR\_BGR2GRAY)

return final\_image.astype(np.uint8)

imList = glob.glob('./raw/\*')

for file in imList:

str = os.path.split(file)

index = str[-1][9:-4] # Take file number

if index == "1":

raw = read\_transparent\_png(file)

\_, bw = cv2.threshold(raw,254,255,cv2.THRESH\_BINARY\_INV)

nz = np.nonzero(bw)

box = [np.min(nz[0]), np.max(nz[0]), np.min(nz[1]), np.max(nz[1])]

print(box)

img = raw[box[0]:box[1],box[2]:box[3]]

"""contours = findContours(img, cv2.RETR\_LIST, cv2.CHAIN\_APPROX\_SIMPLE)

for i, c in enumerate(contours):

area = cv2.contourArea(c)

areaArray.append(area)"""

"""

plt.imshow(img, cmap='gray', interpolation = 'bicubic')

plt.xticks([]), plt.yticks([]) # to hide tick values on X and Y axis

plt.show()"""

cv2.imwrite("./crop/cropped{}.png".format(index), img)

break

**Resize Imgaes**

import cv2

import glob

import os

imList = glob.glob('./crop/\*')

for file in imList:

str = os.path.split(file)

index = str[-1][7:-4] # Take file number

if True:

raw = cv2.imread(file)

img = cv2.resize(raw, (320, 240), cv2.INTER\_CUBIC)

cv2.imwrite("./crop240/resized{}.png".format(index), img)

**OB v.2 Programme Codes**

**Main.py**

# TODO: Bu satirlar silinecek

#from subprocess import call

#call(["pyside2-uic", "/home/volkan/proje/qt/ob2/ob2/ob2.ui", "-o", "/home/volkan/proje/qt/ob2/ob2/ob2.py"])

import sys

from PySide2.QtWidgets import QApplication, QMainWindow, QFrame, QHBoxLayout, QVBoxLayout, QLabel

from PySide2.QtCore import Slot, Signal, Qt

from PySide2.QtGui import QPixmap

import ob2

import cv2

import os

import numpy as np

import random

import math

class ClickableLabel(QLabel):

clicked = Signal()

def \_\_init\_\_(self, parent):

super(ClickableLabel, self).\_\_init\_\_()

self.setParent(parent)

def mousePressEvent(self, event):

self.clicked.emit()

class OB2(QMainWindow, ob2.Ui\_MainWindow):

def \_\_init\_\_(self):

QMainWindow.\_\_init\_\_(self)

self.setupUi(self)

self.path = os.path.dirname(os.path.abspath(\_\_file\_\_))

self.shapes = np.array(list(range(1, 21))+list(range(26, 32))+list(range(37, 41))+[62, 63, 70, 71])

self.pushButton.clicked.connect(self.create\_images)

self.no\_rows = math.ceil(len(self.shapes)/3)

self.no\_img = len(self.shapes)

self.shapes\_selected = np.zeros(self.no\_img, dtype='uint8')

self.img\_rows = np.empty((1, self.no\_rows), dtype=QFrame)

self.img\_rows\_lay = np.empty((1, self.no\_rows), dtype=QHBoxLayout)

self.img\_labels = np.empty((self.no\_rows, 3), dtype=ClickableLabel)

self.init\_images()

self.random = 1

self.scrollAreaWidgetContents.setEnabled(False)

self.radioButton.clicked.connect(self.select\_random)

self.radioButton\_2.clicked.connect(self.select\_manual)

def select\_random(self):

self.radioButton.setChecked(True)

self.radioButton\_2.setChecked(False)

self.random = 1

self.scrollAreaWidgetContents.setEnabled(False)

def select\_manual(self):

self.radioButton.setChecked(False)

self.radioButton\_2.setChecked(True)

self.random = 0

self.scrollAreaWidgetContents.setEnabled(True)

def init\_images(self):

folder\_in = 'INPUT\_MAIN'

for i in range(self.no\_rows):

self.img\_rows[0, i] = QFrame(self.scrollAreaWidgetContents)

self.img\_rows\_lay[0, i] = QHBoxLayout(self.img\_rows[0, i])

self.verticalLayout\_3.addWidget(self.img\_rows[0, i])

for j in range(3):

if (i\*3+j) < self.no\_img:

self.img\_labels[i, j] = ClickableLabel(self.img\_rows[0, i])

self.img\_labels[i, j].setAlignment(Qt.AlignHCenter)

self.img\_rows\_lay[0, i].addWidget(self.img\_labels[i, j])

pixmap = QPixmap("{}/{}/input{}.png".format(self.path, folder\_in, self.shapes[(i\*3+j)]))

self.img\_labels[i, j].setPixmap(pixmap)

self.img\_labels[i, j].setLineWidth(10)

self.assign(i, j)

def assign(self, i, j):

self.img\_labels[i, j].clicked.connect(lambda: self.img\_click(i, j))

@Slot()

def img\_click(self, n, m):

if self.shapes\_selected[3\*n+m] == 0:

self.img\_labels[n, m].setFrameShape(QFrame.Shape(1))

self.shapes\_selected[3\*n+m] = 1

elif self.shapes\_selected[3\*n+m] == 1:

self.img\_labels[n, m].setFrameShape(QFrame.Shape(0))

self.shapes\_selected[3\*n+m] = 0

def find\_alpha\_comp (self, alpha, beta):

return alpha + np.multiply(beta, 1-alpha).astype('float32')

def find\_color\_comp (self, ca, cb, aa, ab):

trans\_mat = self.find\_alpha\_comp(aa, ab)

nom = np.multiply(ca, aa) + np.multiply(np.multiply(cb, ab), (1 - aa))

return np.divide(nom, trans\_mat, out=np.zeros\_like(nom), where=trans\_mat!=0)

def showim(self, img):

cv2.imshow("Test",img)

cv2.waitKey(0)

@Slot()

def create\_images(self):

folder = 'FINAL\_OB2'

folder\_in = 'INPUT\_MAIN'

folder\_neg = 'FINAL\_NEG'

folder\_res = 'FINAL\_RESULT'

kernel = np.ones((7, 7), dtype='float32')

cl = cv2.createCLAHE(2, (8, 8))

if self.random == 1:

shape\_list = self.shapes

shape\_list = np.sort((np.random.choice(shape\_list, np.min([self.no\_img, int(self.lineEdit.text())]) , False)))

elif self.random == 0:

shape\_list = self.shapes[self.shapes\_selected == 1]

file\_list = []

for shape in shape\_list:

idx = random.choice(range(1, 21))

file\_list.append('{}\_{}'.format(shape, idx))

white = np.ones((3508, 4961, 4), dtype='float32')

white[:, :, 3] = 0

white\_in = np.ones((3508, 4961, 4), dtype='float32')

white\_in[:, :, 3] = 0

white\_neg = np.ones((3508, 4961, 4), dtype='float32')

white\_neg[:, :, 3] = 0

background = np.ones((3508, 4961), dtype='float32')

for ind, file in enumerate(file\_list):

IMG\_SIZE = np.array([1, 1]) \* np.random.randint(750, 2000)

all\_255 = np.ones(IMG\_SIZE, dtype='float32')

pt = (np.random.randint(0, 4960 - IMG\_SIZE[1]), np.random.randint(0, 3507 - IMG\_SIZE[0]))

raw\_in = cv2.imread("{}/{}/input{}.png".format(self.path, folder\_in, shape\_list[ind]))

raw\_neg = cv2.imread("{}/{}/res{}.png".format(self.path, folder\_neg, file))

raw = cv2.imread("{}/{}/res{}.png".format(self.path, folder, file))

img\_in = cv2.cvtColor(raw\_in, cv2.COLOR\_BGR2GRAY)

img\_in = cv2.resize(img\_in, (IMG\_SIZE[0], IMG\_SIZE[1]), cv2.INTER\_CUBIC)

img\_in = (img\_in / 255.0)

img\_neg = cv2.cvtColor(raw\_neg, cv2.COLOR\_BGR2GRAY)

img\_neg = cv2.resize(img\_neg, (IMG\_SIZE[0], IMG\_SIZE[1]), cv2.INTER\_CUBIC)

img\_neg = (img\_neg / 255.0)

img = cv2.cvtColor(raw, cv2.COLOR\_BGR2GRAY)

img = cl.apply(img)

img = cv2.resize(img, (IMG\_SIZE[0], IMG\_SIZE[1]), cv2.INTER\_CUBIC)

img = cv2.morphologyEx(img, cv2.MORPH\_OPEN, kernel)

img = cv2.morphologyEx(img, cv2.MORPH\_CLOSE, kernel)

img = (img/255.0)

bright = (np.random.rand() + 1.5) # Random saturation

neg\_in = (1 - img\_in)

neg\_in = np.minimum(neg\_in, all\_255).astype('float32')

neg\_neg = (1 - img\_neg)\*bright

neg\_neg = np.minimum(neg\_neg, all\_255).astype('float32')

neg = (1 - img)\*bright

neg = np.minimum(neg, all\_255).astype('float32')

mask\_in = np.array(neg\_in)

mask\_in[mask\_in != 0] = 1

clipped\_in = white\_in[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], :]

mask\_neg = np.array(neg\_neg)

mask\_neg[mask\_neg != 0] = 1

clipped\_neg = white\_neg[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], :]

mask = np.array(neg)

mask[mask != 0] = 1

clipped = white[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], :]

chroma = np.random.rand(3)

ch\_r\_in = self.find\_color\_comp(mask\_in \* chroma[0], clipped\_in[:, :, 2], neg\_in, clipped\_in[:, :, 3])

ch\_g\_in = self.find\_color\_comp(mask\_in \* chroma[1], clipped\_in[:, :, 1], neg\_in, clipped\_in[:, :, 3])

ch\_b\_in = self.find\_color\_comp(mask\_in \* chroma[2], clipped\_in[:, :, 0], neg\_in, clipped\_in[:, :, 3])

ch\_a\_in = self.find\_alpha\_comp(neg\_in, clipped\_in[:, :, 3])

ch\_r\_neg = self.find\_color\_comp(mask\_neg \* chroma[0], clipped\_neg[:, :, 2], neg\_neg, clipped\_neg[:, :, 3])

ch\_g\_neg = self.find\_color\_comp(mask\_neg \* chroma[1], clipped\_neg[:, :, 1], neg\_neg, clipped\_neg[:, :, 3])

ch\_b\_neg = self.find\_color\_comp(mask\_neg \* chroma[2], clipped\_neg[:, :, 0], neg\_neg, clipped\_neg[:, :, 3])

ch\_a\_neg = self.find\_alpha\_comp(neg\_neg, clipped\_neg[:, :, 3])

ch\_r = self.find\_color\_comp(mask\* chroma[0], clipped[:, :, 2], neg, clipped[:, :, 3])

ch\_g = self.find\_color\_comp(mask\* chroma[1], clipped[:, :, 1], neg, clipped[:, :, 3])

ch\_b = self.find\_color\_comp(mask\* chroma[2], clipped[:, :, 0], neg, clipped[:, :, 3])

ch\_a = self.find\_alpha\_comp(neg, clipped[:, :, 3])

white\_in[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 0] = ch\_b\_in

white\_in[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 1] = ch\_g\_in

white\_in[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 2] = ch\_r\_in

white\_in[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 3] = ch\_a\_in

white\_neg[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 0] = ch\_b\_neg

white\_neg[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 1] = ch\_g\_neg

white\_neg[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 2] = ch\_r\_neg

white\_neg[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 3] = ch\_a\_neg

white[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 0] = ch\_b

white[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 1] = ch\_g

white[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 2] = ch\_r

white[pt[1]:pt[1] + IMG\_SIZE[1], pt[0]:pt[0] + IMG\_SIZE[0], 3] = ch\_a

chroma = np.random.rand(3)

white\_in[:, :, 2] = self.find\_color\_comp(white\_in[:, :, 2], background \* chroma[0], white\_in[:, :, 3], background)

white\_in[:, :, 1] = self.find\_color\_comp(white\_in[:, :, 1], background \* chroma[1], white\_in[:, :, 3], background)

white\_in[:, :, 0] = self.find\_color\_comp(white\_in[:, :, 0], background \* chroma[2], white\_in[:, :, 3], background)

white\_in[:, :, 3] = background

white\_neg[:, :, 2] = self.find\_color\_comp(white\_neg[:, :, 2], background \* chroma[0], white\_neg[:, :, 3], background)

white\_neg[:, :, 1] = self.find\_color\_comp(white\_neg[:, :, 1], background \* chroma[1], white\_neg[:, :, 3], background)

white\_neg[:, :, 0] = self.find\_color\_comp(white\_neg[:, :, 0], background \* chroma[2], white\_neg[:, :, 3], background)

white\_neg[:, :, 3] = background

white[:, :, 2] = self.find\_color\_comp(white[:, :, 2], background \* chroma[0], white[:, :, 3], background)

white[:, :, 1] = self.find\_color\_comp(white[:, :, 1], background \* chroma[1], white[:, :, 3], background)

white[:, :, 0] = self.find\_color\_comp(white[:, :, 0], background \* chroma[2], white[:, :, 3], background)

white[:, :, 3] = background

white = 255 \* white

white\_in = 255 \* white\_in

white\_neg = 255 \* white\_neg

out\_no = math.floor((len(os.listdir("{}/{}".format(self.path, folder\_res))) / 3)+1)

cv2.imwrite("{}/{}/final\_out{}.png".format(self.path, folder\_res, out\_no), white)

cv2.imwrite("{}/{}/final\_neg{}.png".format(self.path, folder\_res, out\_no), white\_neg)

cv2.imwrite("{}/{}/final\_in{}.png".format(self.path, folder\_res, out\_no), white\_in)

if \_\_name\_\_ == "\_\_main\_\_":

app = QApplication([])

window = OB2()

window.show()

sys.exit(app.exec\_())

**Ob2.py**

# -\*- coding: utf-8 -\*-

################################################################################

## Form generated from reading UI file 'ob2.ui'

##

## Created by: Qt User Interface Compiler version 5.14.2

##

## WARNING! All changes made in this file will be lost when recompiling UI file!

################################################################################

from PySide2.QtCore import (QCoreApplication, QDate, QDateTime, QMetaObject,

QObject, QPoint, QRect, QSize, QTime, QUrl, Qt)

from PySide2.QtGui import (QBrush, QColor, QConicalGradient, QCursor, QFont,

QFontDatabase, QIcon, QKeySequence, QLinearGradient, QPalette, QPainter,

QPixmap, QRadialGradient)

from PySide2.QtWidgets import \*

class Ui\_MainWindow(object):

def setupUi(self, MainWindow):

if not MainWindow.objectName():

MainWindow.setObjectName(u"MainWindow")

MainWindow.resize(640, 480)

self.centralwidget = QWidget(MainWindow)

self.centralwidget.setObjectName(u"centralwidget")

self.verticalLayout = QVBoxLayout(self.centralwidget)

self.verticalLayout.setObjectName(u"verticalLayout")

self.scrollArea = QScrollArea(self.centralwidget)

self.scrollArea.setObjectName(u"scrollArea")

self.scrollArea.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOn)

self.scrollArea.setWidgetResizable(True)

self.scrollAreaWidgetContents = QWidget()

self.scrollAreaWidgetContents.setObjectName(u"scrollAreaWidgetContents")

self.scrollAreaWidgetContents.setGeometry(QRect(0, 0, 606, 203))

self.verticalLayout\_3 = QVBoxLayout(self.scrollAreaWidgetContents)

self.verticalLayout\_3.setObjectName(u"verticalLayout\_3")

self.scrollArea.setWidget(self.scrollAreaWidgetContents)

self.verticalLayout.addWidget(self.scrollArea)

self.groupBox = QGroupBox(self.centralwidget)

self.groupBox.setObjectName(u"groupBox")

self.verticalLayout\_2 = QVBoxLayout(self.groupBox)

self.verticalLayout\_2.setObjectName(u"verticalLayout\_2")

self.frame = QFrame(self.groupBox)

self.frame.setObjectName(u"frame")

self.frame.setLayoutDirection(Qt.LeftToRight)

self.frame.setFrameShape(QFrame.NoFrame)

self.frame.setFrameShadow(QFrame.Raised)

self.horizontalLayout = QHBoxLayout(self.frame)

self.horizontalLayout.setObjectName(u"horizontalLayout")

self.radioButton = QRadioButton(self.frame)

self.radioButton.setObjectName(u"radioButton")

sizePolicy = QSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)

sizePolicy.setHorizontalStretch(0)

sizePolicy.setVerticalStretch(0)

sizePolicy.setHeightForWidth(self.radioButton.sizePolicy().hasHeightForWidth())

self.radioButton.setSizePolicy(sizePolicy)

self.radioButton.setMinimumSize(QSize(200, 0))

self.radioButton.setChecked(True)

self.horizontalLayout.addWidget(self.radioButton)

self.lineEdit = QLineEdit(self.frame)

self.lineEdit.setObjectName(u"lineEdit")

sizePolicy.setHeightForWidth(self.lineEdit.sizePolicy().hasHeightForWidth())

self.lineEdit.setSizePolicy(sizePolicy)

self.lineEdit.setAlignment(Qt.AlignCenter)

self.horizontalLayout.addWidget(self.lineEdit)

self.horizontalSpacer = QSpacerItem(40, 20, QSizePolicy.Expanding, QSizePolicy.Minimum)

self.horizontalLayout.addItem(self.horizontalSpacer)

self.verticalLayout\_2.addWidget(self.frame)

self.frame\_2 = QFrame(self.groupBox)

self.frame\_2.setObjectName(u"frame\_2")

self.frame\_2.setFrameShape(QFrame.NoFrame)

self.frame\_2.setFrameShadow(QFrame.Raised)

self.horizontalLayout\_2 = QHBoxLayout(self.frame\_2)

self.horizontalLayout\_2.setObjectName(u"horizontalLayout\_2")

self.radioButton\_2 = QRadioButton(self.frame\_2)

self.radioButton\_2.setObjectName(u"radioButton\_2")

self.horizontalLayout\_2.addWidget(self.radioButton\_2)

self.verticalLayout\_2.addWidget(self.frame\_2)

self.pushButton = QPushButton(self.groupBox)

self.pushButton.setObjectName(u"pushButton")

self.pushButton.setMinimumSize(QSize(0, 50))

self.verticalLayout\_2.addWidget(self.pushButton)

self.verticalLayout.addWidget(self.groupBox)

MainWindow.setCentralWidget(self.centralwidget)

self.menubar = QMenuBar(MainWindow)

self.menubar.setObjectName(u"menubar")

self.menubar.setGeometry(QRect(0, 0, 640, 22))

MainWindow.setMenuBar(self.menubar)

self.statusbar = QStatusBar(MainWindow)

self.statusbar.setObjectName(u"statusbar")

MainWindow.setStatusBar(self.statusbar)

self.toolBar = QToolBar(MainWindow)

self.toolBar.setObjectName(u"toolBar")

MainWindow.addToolBar(Qt.TopToolBarArea, self.toolBar)

self.toolBar\_2 = QToolBar(MainWindow)

self.toolBar\_2.setObjectName(u"toolBar\_2")

MainWindow.addToolBar(Qt.TopToolBarArea, self.toolBar\_2)

self.toolBar\_3 = QToolBar(MainWindow)

self.toolBar\_3.setObjectName(u"toolBar\_3")

MainWindow.addToolBar(Qt.TopToolBarArea, self.toolBar\_3)

self.retranslateUi(MainWindow)

QMetaObject.connectSlotsByName(MainWindow)

# setupUi

def retranslateUi(self, MainWindow):

MainWindow.setWindowTitle(QCoreApplication.translate("MainWindow", u"OB2 G\u00f6rsel Olu\u015fturma Arac\u0131", None))

self.groupBox.setTitle(QCoreApplication.translate("MainWindow", u"Se\u00e7im Tipi", None))

self.radioButton.setText(QCoreApplication.translate("MainWindow", u"Rastgele Se\u00e7im", None))

self.lineEdit.setText(QCoreApplication.translate("MainWindow", u"20", None))

self.radioButton\_2.setText(QCoreApplication.translate("MainWindow", u"Elle Se\u00e7im", None))

self.pushButton.setText(QCoreApplication.translate("MainWindow", u"Olu\u015ftur !", None))

self.toolBar.setWindowTitle(QCoreApplication.translate("MainWindow", u"toolBar", None))

self.toolBar\_2.setWindowTitle(QCoreApplication.translate("MainWindow", u"toolBar\_2", None))

self.toolBar\_3.setWindowTitle(QCoreApplication.translate("MainWindow", u"toolBar\_3", None))

# retranslateUi