Sample Code for 'Dermoscopic Image Classification with Neural Style Transfer'

0 Introduction

This is a sample code to generate the stylized images and reproduce the classification results of the simulated data set, i.e., the constructed data set of 1000 images in the manuscript "Dermoscopic Image Classification with Neural Style Transfer". The data set can be downloaded at **'cralo31/dermoST/'** on Github. The data folder readily provides the original and pre-processed lesion images, along with the generated segmentation masks (rotated and centered versions are also provided for ABCD calculation and reference purposes) of the 1000 images. The github repository also includes the VGG weights used in this paper. The dermoscopic images considered in this paper are also available for public download through the ISIC database (https://www.isic-archive.com/#!/topWithHeader/wideContentTop/main).

Both Python and R code are used in the analysis. To directly illustrate our method, we first present the code of the proposed method, followed by the feature extraction (CP decomposition and ABCD features) and classification. Pre-processing of the images and the generation of the segmentation masks are also provided for reference. Our code is presented in the following order.

Main

1. Proposed Style Transfer Algorithm for Skin Lesions (Python)
2. Feature Extraction (R)
3. Classification (R)

Appendix

4. Pre-processing (Python)
5. Lesion Segmentation (Python)

0.1 Dataset

The data set folder contains 6 sub folders:

1. Original_Images: 1000 dermoscopic images selected from the ISIC database and resized to 224-by-224 pixels. Image artifacts such as hair and air bubbles are removed.

2. Color_Normalized_Images: The Shades of Gray algorithm is applied to the original images for color and illumination normalization. These images are regarded as the "raw images" in the manuscript.

3. Content_Image: This folder contains a single content image, which is constructed by taking the average of the 3 RGB channels across all the images in Original_Images. This is the content image used throughout the style transfer process.

4. Segmentation_Masks: This folder contains 1000 segmentation masks generated by U-net which is trained on the PH2 data set.

5. Segmentation_Masks: This folder contains 1000 segmentation masks generated that is centered and rotated along its principal axis.

6. Stylized_Images: This folder contains 1000 generated images that yielded the best classification performance using the proposed style-transfer pipeline.

0.2 Load in libraries

Load in the libraries for Python and R respectively. These need to be installed beforehand.

```
from __future__ import print_function
import tensorflow as tf
from keras.preprocessing.image import load_img, save_img, img_to_array
from keras.backend.tensorflow_backend import set_session,
clear_session, get_session
import numpy as np
from scipy.optimize import fmin_l_bfgs_b
import time
import argparse
import os
import gc
from tensorflow.keras.applications import vgg19
from keras import backend as K
import glob
import sys, math
from PIL import Image
from scipy import ndimage
import os, glob, time, re, PIL, datetime, random
from numpy import asarray
import pandas as pd
from PIL import Image
from os import listdir
from matplotlib import image
from sklearn import datasets, linear_model
from sklearn.model_selection import train_test_split, KFold,
RepeatedKFold, StratifiedKFold
from matplotlib import pyplot as plt
```

```
import matplotlib.image as mpimg
import cv2
from scipy import stats
from tqdm import tqdm_notebook, tnrange
from itertools import chain
from skimage.io import imread, imshow, concatenate_images
from skimage.transform import resize
from skimage.morphology import label
from sklearn.model_selection import train_test_split
from PIL import Image, ImageOps
import skimage

# Download the package 'tnsrcomp' and 'MatrixFact' from author's Github
install.packages("devtools")  # Need 'devtools' package for Github
installation
devtools::install_github('cralo31/tnsrcomp')    # tnsrcomp
devtools::install_github('cralo31/MatrixFact')  # MatrixFact

# Load in packages #
pkgs = c("reticulate", "Rcpp", "RcppArmadillo", "irlba", "MatrixFact",
         "pracma", "tibble", "glmnet","randomForest", "caret", "Hmisc",
"abind",
         "magick", "imager","keras", "tensorflow", "stringr",
"rTensor", "doRNG",
         "ROCR","reticulate","class","keras", "e1071", "nnTensor",
"devtools", "doParallel",
         "tnsrcomp", "RcppEigen", "inline", "microbenchmark",
"pheatmap", "imager",
         "DescTools", "ROCR",  "OpenImageR")
lapply(pkgs, require, character.only = TRUE)
```

1 Proposed Neural Style Transfer Algorithm for Skin Lesions

This section presents the style transfer algorithm and an example of running the algorithm.

1.1 Generate an Example Image

```
### First read the file names in a folder and sort them by numeric
order
def readData(inDir, mode):

    # Read in the files
    regex = re.compile(r'\d+')
    imgfiles = glob.glob(inDir + '/*.jpg')
    imgfiles = [f for f in glob.glob(inDir + "**/*.jpg", recursive =
True)]

    # Extract image name and sort in ascending order
```

```
    if mode == 1:
        imgnum = [0] * len(imgfiles)
        for i in range(0, len(imgfiles)):
            imgnum[i] = [int(x) for x in regex.findall(imgfiles[i])][0]
        filenum = pd.DataFrame(list(zip(imgfiles, imgnum)), columns =
['files','index'])
        filenum = filenum.sort_values('index')
        filenum = filenum.set_index('index')
        return filenum['files'].to_numpy()

    return imgfiles


# Load the style, content images and the mask of each style image
sty_images = readData('./dermoST_data/Color_Normalized_Images/', 1)
sty_masks = readData('./dermoST_data/Segmentation_Masks/', 1)
cont_image = readData('./dermoST_data/Content_Image/', 2)

###!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!###
# Generate a single image (any parameters not specified is default
according
# to 'main_NST' function)
newimage = main_NST(hard_width = 224, iteration = 500,
                    output_dir = './**output_folder**/',
                    content_img = cont_image, style_img =
sty_images[0],
                    style_mask = sty_masks[0],
                    content_weight = 1, style_weight = 100, tv_weight =
1)
###!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!###
```

1.2 Setup and component functions (loss, read images, etc.)

```
#########################################################################
########
### Masked Neural Style Transfer Algorithm ###

# Load in the VGG weights
vgg_weights = prepare_model('./imagenet-vgg-verydeep-19.mat')

# Wrapper function to generate an image according to input parameters
# Tuning parameters are self-explanatory
def main_NST(content_layers=['relu4_1'],
          content_layers_weights=[1.0],
          content_loss_normalization=1,
          content_weight=1.0,
          feature_pooling_type='max',
          hard_width=None,
```

```
          init_noise_ratio=0.0,
          iteration=1000,
          learning_rate=10.0,
          log_iteration=10,
          mask_downsample_type='simple',
          mask_n_colors=1,
          mask_normalization_type='square_sum',
          model_path='imagenet-vgg-verydeep-19.mat',
          optimizer='lbfgs',
          output_dir='./output',
          content_img=None,
          style_img=None,
          style_layers=['relu1_1', 'relu2_1', 'relu3_1', 'relu4_1',
'relu5_1'],
          style_layers_weights=[1.0, 1.0, 1.0, 1.0, 1.0],
          style_mask=None, style_weight=0.2,
          target_mask=None, tv_weight=0.0):

    '''
    init
    '''

    reset_keras()

    # Read in the features
    style_image = read_image(style_img, hard_width)
    content_image = read_image(content_img, hard_width)

    # Style features
    if len(style_layers) == 1:
        style_layers_weights = [1.0]
    else:
        style_layers_weights=[1.0, 1.0, 1.0, 1.0, 1.0]

    style_features = compute_features(vgg_weights, 'max', style_image,
style_layers)
    content_features = compute_features(vgg_weights, 'max',
content_image, content_layers)

    # Load in the target mask
    target_masks_origin = read_single_mask(target_mask, hard_width)
    style_masks_origin = read_single_mask(style_mask, hard_width)

    # Generate the guidance channel
    #reset_keras()
    target_masks = compute_layer_masks(target_masks_origin,
style_layers, 'simple')
    style_masks = compute_layer_masks(style_masks_origin, style_layers,
'simple')
```

```
    # init img & target shape
    target_shape = content_image.shape
    init_img = get_init_image(content_image, init_noise_ratio)

    print("features loaded")

    target_net = build_target_net(vgg_weights, feature_pooling_type,
                                  (1, hard_width, hard_width, 3))

    '''
    Loss
    '''

    content_loss = sum_content_loss(target_net, content_features,
                                    content_layers,
content_layers_weights,
                                    content_loss_normalization)

    style_masked_loss = sum_masked_style_loss(target_net,
style_features,
                                              target_masks,
style_masks,
                                              style_layers,
style_layers_weights,
                                              mask_normalization_type)

    tv_loss = sum_total_variation_loss(target_net['input'],
target_shape)

    total_loss = content_weight * content_loss + \
                 style_weight * style_masked_loss + \
                 tv_weight * tv_loss

    print('Loss')
    '''
    Optimizer
    '''
    optimizer = ScipyOptimizerInterface(total_loss, method='L-BFGS-B',
                                        options={'maxiter': iteration,
                                        'disp': log_iteration,
'ftol':0.0005})
    # init
    init_op = tf.global_variables_initializer()
    sess = tf.Session()
    sess.run(init_op)
    sess.run(target_net['input'].assign(init_img) )
    # train
    optimizer.minimize(sess)
```

```
    print('Optimized')

    '''
    Output
    '''

    result = sess.run(target_net['input'])

    return result[0]

######################################################################
################
### Individual Component Functions for Masked NST Algorithm ###

# Read a lesion image and convert it into a vector (normalized)
def read_image(path, hard_width):   # read and preprocess
    img = Image.open(path)
    if hard_width:
        img = img.resize((hard_width, hard_width))
    img = np.array(img)
    img = img.astype(np.float32)
    img = img[np.newaxis, :, :, :]
    img = img - [123.68, 116.779, 103.939]
    return img

# Read a lesion mask and convert it into a tensor (normalized)
def read_single_mask(path, hard_width):
    rawmask = Image.open(path)
    if hard_width:
        rawmask = rawmask.resize((hard_width, hard_width))
    rawmask = np.array(rawmask)
    rawmask = rawmask / 255 # integer division, only pure white pixels
become 1
    rawmask = rawmask.astype(np.float32)
    rawmask = rawmask[np.newaxis, :, :]

    return np.stack(rawmask)

def write_image(path, img):   # Postprocess image and output
    img = img + [123.68, 116.779, 103.939]
    img = img[0]
    img = np.clip(img, 0, 255).astype('uint8')
    scipy.misc.imsave(path, img)

# Initialize the image
def get_init_image(content_img, init_noise_ratio):
    noise_img = np.random.uniform(-20., 20.,
content_img.shape).astype(np.float32)
```

```
    init_img = init_noise_ratio * noise_img + (1. - init_noise_ratio) *
content_img
    return init_img


'''
    compute features & masks
    build net
'''
# Define the architecture of the VGG model
vgg_layers = (
    'conv1_1', 'relu1_1', 'conv1_2', 'relu1_2', 'pool1',

    'conv2_1', 'relu2_1', 'conv2_2', 'relu2_2', 'pool2',

    'conv3_1', 'relu3_1', 'conv3_2', 'relu3_2', 'conv3_3',
    'relu3_3', 'conv3_4', 'relu3_4', 'pool3',

    'conv4_1', 'relu4_1', 'conv4_2', 'relu4_2', 'conv4_3',
    'relu4_3', 'conv4_4', 'relu4_4', 'pool4',

    'conv5_1', 'relu5_1', 'conv5_2', 'relu5_2', 'conv5_3',
    'relu5_3', 'conv5_4', 'relu5_4'
)

# Prepare the raw vgg model
def prepare_model(path):
    vgg_rawnet = scipy.io.loadmat(path)
    return vgg_rawnet['layers'][0] # another solution: global
vgg_weights

# Construct a convolution layer
def conv_layer(input, W, b):
    conv =  tf.nn.conv2d(input, W, strides=[1,1,1,1], padding='SAME')
    return conv + b

# Construct a pooling layer
def pool_layer(input, feature_pooling_type):
    if feature_pooling_type == 'avg':
        return tf.nn.avg_pool(input, ksize=[1, 2, 2, 1],
      strides=[1, 2, 2, 1], padding='SAME')
    elif feature_pooling_type == 'max':
        return tf.nn.max_pool(input, ksize=[1, 2, 2, 1],
      strides=[1, 2, 2, 1], padding='SAME')

# Construct an entire imagenet model with the specified parameters
def build_image_net(input_tensor, vgg_weights, feature_pooling_type):
    net = {}
    current = input_tensor
```

```
    for i, name in enumerate(vgg_layers):
        layer_kind = name[:4]
        if layer_kind == 'conv':
            weights, bias = vgg_weights[i][0][0][2][0]
            bias = bias.reshape(-1)
            current = conv_layer(current, tf.constant(weights),
tf.constant(bias))
        elif layer_kind == 'relu':
            current = tf.nn.relu(current)
        elif layer_kind == 'pool':
            current = pool_layer(current, feature_pooling_type)
        net[name] = current

    return net

# Construct the vgg network of the generated image
def build_target_net(vgg_weights, pooling_type, target_shape):
    input = tf.Variable( np.zeros(target_shape).astype('float32') )
    net = build_image_net(input, vgg_weights, pooling_type)
    net['input'] = input
    return net

# Skeleton network to down sample masks (nothing is trained)
def build_mask_net(input_tensor, mask_downsample_type):
    net = {}
    current = input_tensor

    # soft
    if mask_downsample_type == 'simple':
        for name in vgg_layers:
            layer_kind = name[:4]
            if layer_kind == 'pool':
                current = tf.nn.avg_pool(current, ksize=[1,2,2,1],
                strides=[1,2,2,1], padding='SAME')
            net[name] = current

    # hard
    elif mask_downsample_type == 'all':
        for name in vgg_layers:
            layer_kind = name[:4]
            if layer_kind == 'conv':
                current = tf.nn.max_pool(current, ksize=[1,3,3,1],
                strides=[1,1,1,1], padding='SAME')
            elif layer_kind == 'pool':
                current = tf.nn.max_pool(current, ksize=[1,2,2,1],
                strides=[1,2,2,1], padding='SAME')
            net[name] = current
```

```
    # hard, keep the padding boundary unchanged
    elif mask_downsample_type == 'inside':
        current = 1 - current
        for name in vgg_layers:
            layer_kind = name[:4]
            if layer_kind == 'conv':
                current = tf.nn.max_pool(current, ksize=[1,3,3,1],
                strides=[1,1,1,1], padding='SAME')
            elif layer_kind == 'pool':
                current = tf.nn.max_pool(current, ksize=[1,2,2,1],
                strides=[1,2,2,1], padding='SAME')
            net[name] = 1 - current

    # soft
    elif mask_downsample_type == 'mean':
        for name in vgg_layers:
            layer_kind = name[:4]
            if layer_kind == 'conv':
                current = tf.nn.avg_pool(current, ksize=[1,3,3,1],
                strides=[1,1,1,1], padding='SAME')
            elif layer_kind == 'pool':
                current = tf.nn.avg_pool(current, ksize=[1,2,2,1],
                strides=[1,2,2,1], padding='SAME')
            net[name] = current

    return net

def compute_features(vgg_weights, pooling_type, input_img, layers):
    input = tf.placeholder(tf.float32, shape=input_img.shape)
    net = build_image_net(input, vgg_weights, pooling_type)
    features = {}
    with tf.Session() as sess:
        for layer in layers:
            features[layer] = sess.run(net[layer], feed_dict={input:
input_img})
    return features

def compute_layer_masks(masks, layers, ds_type):
    masks_tf = masks.transpose([1,2,0]) # [numberOfMasks, h, w] -> [h,
w, masks]
    masks_tf = masks_tf[np.newaxis, :, :, :] # -> [1, h, w, masks]

    input = tf.placeholder(tf.float32, shape=masks_tf.shape)
    net = build_mask_net(input, ds_type) # only do pooling, so no
intervention between masks
    layer_masks = {}
    with tf.Session() as sess:
        for layer in layers:
            out = sess.run(net[layer], feed_dict={input: masks_tf})
```

```python
            layer_masks[layer] = out[0].transpose([2,0,1])
    return layer_masks

'''
    loss
'''

# Compute the content loss of each individual layer
def content_layer_loss(p, x, loss_norm):
    _, h, w, d = p.shape
    M = h * w
    N = d
    K = 1. / (N * M)
    if loss_norm  == 1:
        K = 1. / (N * M)
    elif loss_norm == 2:
        K = 1. / (2. * N**0.5 * M**0.5)
    loss = K * tf.reduce_sum( tf.pow((x - p), 2) )
    return loss

# Compute the total content loss
def sum_content_loss(target_net, content_features, layers,
layers_weights, loss_norm):
    content_loss = 0.
    for layer, weight in zip(layers, layers_weights):
        p = content_features[layer]
        x = target_net[layer]
        content_loss += content_layer_loss(p, x, loss_norm) * weight
    content_loss /= float(sum(layers_weights))
    return content_loss

# Masked gram matrix of individual layer
def masked_gram(x, mx, mask_norm, N):
    R = mx.shape[0]
    M = mx.shape[1] * mx.shape[2]

    # TODO: use local variable?
    mx = mx.reshape([R, M])
    x = tf.reshape(x, [M, N])
    x = tf.transpose(x) # N * M
    masked_gram = []
    for i in range(R):
        mask = mx[i]
        masked_x = x * mask
        if mask_norm == 'square_sum':
            norm = 1. / np.sum(mask**2)
        elif mask_norm == 'sum':
            norm = 1. / np.sum(mask)
        gram = norm * tf.matmul(masked_x, tf.transpose(masked_x))
```

```
        masked_gram.append(gram)
    return tf.stack(masked_gram)

# Masked style loss of individual layer
def masked_style_layer_loss(a, ma, x, mx, mask_norm):
    N = a.shape[3]
    R = ma.shape[0]
    norm = 1. / (4. * N**2 * R)
    A = masked_gram(a, ma, mask_norm, N)
    G = masked_gram(x, mx, mask_norm, N)
    loss = norm * tf.reduce_sum( tf.pow((G - A), 2) )
    return loss

# Total masked style loss
def sum_masked_style_loss(target_net, style_features, target_masks,
style_masks, layers, layers_weights, mask_norm):
    style_loss = 0.
    for layer, weight in zip(layers, layers_weights):
        a = style_features[layer]
        ma = style_masks[layer]
        x = target_net[layer]
        mx = target_masks[layer]
        style_loss += masked_style_layer_loss(a, ma, x, mx, mask_norm)
* weight
    style_loss /= float(sum(layers_weights))
    return style_loss

# Compute the gram matrix
def gram_matrix(x):
    _, h, w, d = x.get_shape() # x is a tensor
    M = h.value * w.value
    N = d.value
    F = tf.reshape(x, (M, N))
    G = tf.matmul(tf.transpose(F), F)
    return (1./M) * G

# Total Variation Loss
def sum_total_variation_loss(input, shape):
    b, h, w, d = shape
    x = input
    tv_y_size = b * (h-1) * w * d
    tv_x_size = b * h * (w-1) * d
    loss_y = tf.nn.l2_loss(x[:,1:,:,:] - x[:,:-1,:,:])  # l2_loss() use
1/2 factor
    loss_y /= tv_y_size
    loss_x = tf.nn.l2_loss(x[:,:,1:,:] - x[:,:,:-1,:])
    loss_x /= tv_x_size
    loss = 2 * (loss_y + loss_x)
    loss = tf.cast(loss, tf.float32) # ?
```

```
    return loss

# Reset Keras Session and GPU memory
def reset_keras():
    sess = tf.keras.backend.get_session()
    tf.keras.backend.clear_session()
    sess.close()
    sess = tf.keras.backend.get_session()

    # use the same config as you used to create the session
    config = tf.ConfigProto()
    config.gpu_options.allow_growth=True
    tf.keras.backend.set_session(tf.Session(config=config))
```

2 Feature Extraction

All code in this section is written in R.

This part 1) reads the images into tensors and computes the 2) CP and 3) ABCD features of all the images in the data set.

2.1 Basic Setup

```
#################################################
### Load the images in a folder into a tensor ###

# Read in the images from the directories
readImgs = function(inDir) {

  # Read in the image files
  imgfile = list.files(inDir, full.names = T)
  filedat = cbind(1:length(imgfile), imgfile, rep(0, length(imgfile)))
  colnames(filedat) = c("index", "images", "style")
  for (i in 1:nrow(filedat)) {
    temp.str = str_extract_all(filedat[i,2],"\\(?[0-9,.]+\\)?")[[1]][2]
    filedat[i,3] = as.numeric(temp.str[length(temp.str)])
  }

  # Reorder the files according to the style images to get the index
  filedat = as.data.frame(filedat)
  filedat$index = as.numeric(as.character(filedat$index));
  filedat$images = as.character(filedat$images)
  filedat$style = as.numeric(as.character(filedat$style))
  filedat = filedat[order(filedat$style),]

  return(filedat)
}

# Convert images into a tensor of specified size
```

```r
imgTnsr = function(inDir, imgSize, col, stackChan, gray) {

  # Read in the files
  imgfile = readImgs(inDir)

  # Determine number of channels
  if (gray) {
    chan = 1
  } else {
    chan = dim(load.image(imgfile[1,2]))[4]
  }

  img.tnsr = array(0, dim = c(nrow(imgfile), imgSize, imgSize, chan))
  for (i in 1:nrow(imgfile)) {
    temp = load.image(imgfile[i,2]) %>% resize(imgSize, imgSize)
    if (gray) {
      temp = temp %>% grayscale()
    }
    img.tnsr[i,,,] = temp
  }

  # Stack the three channels together
  if (col) {
    if (stackChan) {
      dim(img.tnsr) = c(nrow(imgfile), imgSize, imgSize * 3)
    }
  }

  return(img.tnsr)
}
```

2.2 Tensor Features

```r
# Load in the training and testing tensor and output the CP features (A
factor matrix) for
# both the training and testing set
basisDC = function(trainChan, testChan, iters, rank) {

  # Unfold the training and testing tensor
  uf_train = unfold_ten(trainChan)
  uf_test = unfold_ten(testChan)
  dimT = dim(trainChan)

  # Perform the tensor decomposition and extract the basis matrices
  trainDC = cp_als(trainChan, uf_train, rank, iter = iters, thres = 1e-
5)

  # Extract the individual factor matrices
  A = trainDC$A; B = trainDC$B; C = trainDC$C
```

```
  # Calculate the test set
  testDC = uf_test$mode1 %*% krao_prod(C,B) %*% pinv(t(krao_prod(C, B))
%*% krao_prod(C, B))

  return(list(trainDC = A, testDC = testDC))
}
```

2.3 ABCD Features

```
###############################################################
### Generate the ABCD Features ###

# Wrapper function to implement the ABCD rule #
abcdFeats = function(imSize) {

  imgFile = "./dermoST_data/Color_Normalized_Images/" # Original lesion
images
  maskFile = "./dermoST_data/Segmentation_Masks/"     # Lesion mask
  rotaFile = "./dermoST_data/PAxis_Rotated_Masks/"    # Masks centered
and rotated to major axis

  # Load in the tensor images
  imgTen = imgTnsr(imgFile, imSize, T, F, F)
  maskTen = imgTnsr(maskFile, imSize, F, F, F); dim(maskTen) =
dim(maskTen)[-4]
  rotaTen = imgTnsr(rotaFile, imSize, F, F, F); dim(rotaTen) =
dim(rotaTen)[-4]

  resMat = matrix(0, 1000, 33)
  for (i in 1:1000) {
    #############################################

    ###For asymmetry, load in the rotated images ###
    rotaMask = binImg(rotaTen[i,,])
    half1 = seq(1, imSize / 2); half2 = seq(imSize / 2 + 1, imSize)

    # Flip the 2nd half 180 degrees and calculate overlapping area #
    # Left/right flip vertical
    lefthalf = rotaMask[half1,]; righthalf = rotaMask[half2,]
    leftflipright = OpenImageR::flipImage(lefthalf, "vertical")
    xlapsum = righthalf + leftflipright
    Xlap = length(which(xlapsum == 2)) / length(which(xlapsum >= 1))

    # Top/bot flip horizontal
    tophalf = rotaMask[,half1]; bothalf = rotaMask[,half2]
    topflipbot = OpenImageR::flipImage(tophalf, "horizontal")
    ylapsum = bothalf + topflipbot
    Ylap = length(which(ylapsum == 2)) / length(which(ylapsum >= 1))

    # Lengthening Index (lengthening and anisotrophy degree)
```

```
    lenMask = momInertia(rotaMask)
    lam1 = 0.5 * (lenMask[5] + lenMask[6] - sqrt((lenMask[5] -
lenMask[6])^2 + 4 * lenMask[4]^2))
    lam2 = 0.5 * (lenMask[5] + lenMask[6] + sqrt((lenMask[5] -
lenMask[6])^2 + 4 * lenMask[4]^2))
    A = lam1 / lam2;

    asymm = c(Xlap, Ylap, A)
    names(asymm) = c("x_symm", "y_symm", "len_index")

    ### Border Irregularity ###
    # Calculate area
    binMask = binImg(maskTen[i,,])
    area = sum(binMask == 1)

    # Calculate perimeter
    # Loop through the image and sum up all pixels that are 1 but has
at least
    # one 0 element in it's 3x3 neighborhood
    peri = 0
    for (j in 2:(imSize-1)) {
      for (k in 2:(imSize-1)) {
        if (binMask[j,k] == 1 & conv(binMask, j, k) != 9) {
          peri = peri + 1
        }
      }
    }
    form = peri^2 / (4 * pi * area)

    ### Color Feats using color table ###
    imgColor = colorSix(i, imgTen, maskTen)

    # Calculate diameter of lesion, use the rotated mask
    diam = finDiam(rotaMask)

    # ABC Feats
    resMat[i,] = c(asymm = asymm, border = form, color = imgColor, diam
= diam)

    if (i == 1) {
      t = c(asymm = asymm, border = form, color = imgColor, diam)
      tname = names(t)
      colnames(resMat) = tname
      colnames(resMat)[c(32,33)] = c("Xdis", "Ydis")
    }
  }

  return(resMat)
}
```

```r
##############################################
# Utility Functions to Compute ABCD features #

# Binarize an image
binImg = function(X){
  X[X > 0.5] = 1; X[X <= 0.5] = 0
  return(X)
}

# Define convolution object
conv = function(image, i, j) {
  vec = c(image[i-1,j+1], image[i, j+1], image[i+1, j+1],
          image[i-1, j], image[i,j], image[i+1, j],image[i-1, j-1],
          image[i, j-1], image[i+1, j-1])
  return(sum(vec))
}

# Function to calculate summary statistics of a single channel #
statFeats = function(chanMat) {
  return (c(summary(c(chanMat)), sd(c(chanMat))))
}

# Function calculate global summmary statistics for each channel of
image #
# Focus on the unmasked (tumor) information only

# Additional mode to consider entire image OR lesion only region
globStats = function(tnsr) {

  stat.mat = matrix(0, dim(tnsr)[1], 7)
  for (i in 1:nrow(tnsr)){
    currImg = tnsr[i,,]
    currImg[currImg < 0.004] = 0
    stat.mat[i,1:7] = statFeats(currImg[currImg != 0])
  }

  return(stat.mat)
}

# Create the lesion color table for each image
colorSix = function(index, imgTen, maskTen) {

  print(index)

  ########################### Color Table
####################################
  # Load the image and corresponding mask
  imPic = imgTen[index,,,]
```

```r
  imMask = maskTen[index,,]

  # Vectorize the pixelated image
  redVec = c(imPic[,,1]); greenVec = c(imPic[,,2]); blueVec =
c(imPic[,,3])
  maskVec = c(imMask)
  pixMat = cbind(redVec, greenVec, blueVec)
  pixMat = pixMat[which(maskVec != 0),]

  # Create pixel distance table
  pixMem = calcMem(pixMat)

  ########################### RGB Statistics
###############################
  rsum = sumstat(pixMat[,1], "R")
  gsum = sumstat(pixMat[,2], "G")
  bsum = sumstat(pixMat[,3], "B")
  rgbstat = c(rsum, gsum, bsum)

  colorstat = c(pixMem, rgbstat)

  return (colorstat)
}

# Function to calculate summary statistics and name the vector
sumstat = function(vec, sting) {

  summ = c(summary(vec), sd(vec))
  names(summ)[7] = "sd"
  names(summ) = paste0(sting, " ", names(summ))

  return(summ)
}


# Create the reference color table
black = rbind(c(0, 62), c(0, 52), c(0, 52))
white = rbind(c(205, 255), c(205, 255), c(205, 255))
red = rbind(c(150, 255), c(0, 52), c(0, 52))
lbrown = rbind(c(150, 240), c(50, 150), c(0, 100))
dbrown = rbind(c(62, 150), c(0, 100), c(0, 100))
bgray = rbind(c(0, 150), c(100, 125), c(125, 150))

colList = list(black = black, white = white, red = red,
               lbrown = lbrown, dbrown = dbrown, bgray = bgray)

for (i in 1:length(colList)) {
  rownames(colList[[i]]) = c("R","G","B")
  colnames(colList[[i]]) = c("min", "max")
```

```
  colList[[i]] = colList[[i]] / 255
}

# Calculate color membership
calcMem = function(pixMat) {

  pixTab = matrix(0, nrow(pixMat), 6)
  colnames(pixTab) = c("black", "white", "red", "lbrown", "dbrown",
"bgray")

  # Calculate color member ship
  for (i in 1:nrow(pixTab)) {

    pix = pixMat[i,]

    # Six ifelse statements to determine color membership
    for (j in 1:length(colList)) {
      curr = colList[[j]]
      if (pix[1] >= curr[1,1] & pix[1] <= curr[1,2] &
          pix[2] >= curr[2,1] & pix[2] <= curr[2,2] &
          pix[3] >= curr[3,1] & pix[3] <= curr[3,2]) {
        pixTab[i,j] = 1
      }
    }
  }

  return(colMeans(pixTab))
}

# Calculate Moment of Inertia
momInertia = function(img, p, q) {

  dims = dim(img)

  # X(colSum) and Y(rowSum)
  xSum = colSums(img)
  ySum = rowSums(img)

  # Calculate moments #

  # Area (Zero Order)
  m00 = sum(img)

  # Centroids (First Order)
  m10 = sum(xSum * 1:dims[1]) / m00
  m01 = sum(ySum * 1:dims[2]) / m00

  # Second Order
  MOI = function(p, q) {
```

```
    pixSum = 0
    for (i in 1:dims[1]) {
      for (j in 1:dims[2]) {
        pixSum = pixSum + (xSum[i]- m10)^p * (ySum[i] - m01)^q
      }
    }
    return(pixSum)
  }
  m11 = MOI(1, 1); m20 = MOI(2, 0); m02 = MOI(0, 2)

  res = c(area = m00, xCen = m10, ycen = m01, m11 = m11, m20 = m20, m02
= m02)

  return(res)
}

# Calculate lesion diameter
# Use the largest box possible to contain the lesion
# Then use the height and width
finDiam = function(matt) {

  # Find uppermost and lowermost points using rowSums
  rsum = rowSums(matt) / dim(matt)[1]
  csum = colSums(matt) / dim(matt)[2]

  # Set threshold to 0.01
  yt = which(rsum > 0.01)
  ydis = abs(yt[1] - yt[length(yt)])

  xt = which(csum > 0.01)
  xdis = abs(xt[1] - xt[length(xt)])

  return(c(xdis, ydis))
}
```

3 Classification

3.1 Classification with CP and ABCD features

```
###!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!###
### Reproduce Classification Results ###

# Load in the raw and style-transferred images directly
rawTensr = imgTnsr("./dermoST_data/Color_Normalized_Images/", 224, T,
F, F)
styTensr = imgTnsr("./dermoST_data/Stylized_Images/", 224, T, F, F)
```

```r
# Compute the ABCD features beforehand
ABCDfeats = abcdFeats(224)

# Compare the result between the raw images vs. stylized images
wrapPar(4, 10, rawTensr, ABCDfeats, 72)   # Raw Images
wrapPar(4, 10, styTensr, ABCDfeats, 72)   # Stylized Images

# Wrapper function to compute classification using CP and ABCD features
#
# with parallel computing and cross-validation #
wrapPar = function(ncores, nfold, tensr, abcdfeats, rank) {

  # Set up the folds for cross validation
  set.seed(1031)
  yResp = factor(c(rep(0, 500), rep(1, 500)))
  cvFolds = createDataPartition(yResp, times = nfold, p = 0.2)

  # Calculate the tensor decomposition and classification in parallel
  cl = makeCluster(ncores)
  registerDoParallel(cl)
  ptm = proc.time()
  run = foreach (foldInd = 1:length(cvFolds),
                 .packages = c("foreach", "Rcpp", "RcppArmadillo",
"doParallel", "doRNG",
                               "tnsrcomp", "caret", "glmnet", "e1071",
"randomForest", "ROCR"),
                 .export = c("trainFeats", "runSup", "calcRes",
"basisDC",
                             "irlba", "knn", "%>%", "cv.glmnet",
"randomForest",
                             "svm", "performance", "prediction",
"pinv", "sensitivity",
                             "specificity"), .noexport = c(),
                 .errorhandling = "pass", .verbose = T) %dorng% {
                   trainFeats(tensr, abcdfeats, foldInd, cvFolds, rank)
                 }
  time = proc.time() - ptm
  stopCluster(cl)

  res = stackRes(run)[[1]]

  return(res)
}

##########################################################################
#######
### Individual Component Function for Classification ###

# Split the data into training/testing split #
```

```
trainFeats = function(tensr, abcdfeats, foldInd, cvFolds, method, rank)
{

  # Load in the actual folds
  fold = cvFolds[[foldInd]]

  # Initialize the response variable
  yResp = factor(c(rep(0, 500), rep(1, 500)))
  yres = list(train = yResp[-fold], test = yResp[fold])

  # ABCD feats
  abcdFeats = list(train = data.frame(abcdfeats[-fold,]), test =
data.frame(abcdfeats[fold,]))
  abcdRes = runSup(abcdFeats$train, abcdFeats$test, yres$train,
yres$test)
  rownames(abcdRes) = paste0("abcd ", rownames(abcdRes))

  #Perform the tensor decomposition
  trainTen = tensr[-fold,,]; testTen = tensr[fold,,]
  dcfeats = basisDC(trainTen, testTen, 30, rank)
  dcTrain = dcfeats$trainDC; dcTest = dcfeats$testDC

  # DC Feats Only
  dcFeatss = list(train = dcTrain, test = dcTest)
  dcRes = runSup(dcFeatss$train, dcFeatss$test, yres$train, yres$test)
  rownames(dcRes) = paste0("dc_", rank, " ", rownames(dcRes))

  # DC + ABCD
  duoFeats = list(train = data.frame(cbind(dcTrain, abcdFeats$train)),
                  test = data.frame(cbind(dcTest, abcdFeats$test)))
  duoRes = runSup(duoFeats$train, duoFeats$test, yres$train, yres$test)
  rownames(duoRes) = paste0("all_", rank, " ", rownames(duoRes))

  return(list(abcd = abcdRes, dcRes = dcRes, allRes = duoRes))
}

# Run supervised learning models on extracted images #
runSup = function(trainX, testX, train.y, test.y) {

  # Scale the training and testing set
  train.x = data.frame(scale(trainX)); test.x =
data.frame(scale(testX))
  train.y = factor(train.y); test.y = factor(test.y)

  # Linear Models #
  alphas = c(0, 0.5, 1);
  lambdas = seq(0.0001, 1, length = 40)
  count = 1
  linearRes = c()
```

```r
  for (i in 1:length(alphas)) {
    for (l in 1:length(lambdas)) {
      ptm <- proc.time()
      linefit = glmnet(as.matrix(train.x), train.y, family =
"binomial", alpha = alphas[i],  lambda = lambdas[l])
      linearRes = rbind(linearRes, calcRes(linefit, test.y, 1, test.x,
(proc.time() - ptm)[3]))
      rownames(linearRes)[count] = paste0("alpha ", alphas[i], "
lambdas ", lambdas[l])
      count = count+1
    }
  }

  # Random Forest
  p = ncol(train.x);
  mtries = unique(c(5, 10, floor(sqrt(p)), floor(p/3)));
  nodes = c(5, 10, 20, 30); rfRes = c();
  count = 1
  for (j in 1:length(mtries)) {
    for (k in 1:length(nodes)) {
      ptm = proc.time()
      rf.fit = randomForest(train.x, train.y, mtry = mtries[j],
nodesize = nodes[k])
      rfRes = rbind(rfRes, calcRes(rf.fit, test.y, 2, test.x,
(proc.time() - ptm)[3]))
      rownames(rfRes)[count] = paste0("mtry ", mtries[j], " node ",
nodes[k])
      count = count + 1
    }
  }

  ### SVM ###
  # Linear Kernel
  cost = 10^(-3:1); svmRes = c();
  for (j in 1:length(cost)) {
    ptm = proc.time()
    svmfit = svm(x = train.x, y = train.y, kernel = "linear", cost =
cost[j], probability = T)
    svmRes = rbind(svmRes, calcRes(svmfit, test.y, 3, test.x,
(proc.time() - ptm)[3]))
    rownames(svmRes)[j] = paste0("cost ", cost[j])
  }

  # Radial Kernel
  gamma = 10^(-3:0); svmRad = c(); count = 1;
  for (j in 1:length(gamma)) {
    for (k in 1:length(cost)) {
      ptm = proc.time()
      svmfit2 = svm(x = train.x, y = train.y, kernel = "radial",
```

```
                    gamma = gamma[j], cost = cost[k], probability = T)
      svmRad = rbind(svmRad, calcRes(svmfit2, test.y, 3, test.x,
(proc.time() - ptm)[3]))
      rownames(svmRad)[count] = paste0("gamma ", gamma[j], " cost ",
cost[k])
      count = count + 1
    }
  }

  # Calculate the classification result
  res = rbind(linearRes, rfRes, svmRes, svmRad)
  colnames(res) = c("Accuracy", "AUC", "Sensitivity", "Specificity",
"Time")

  return(res)
}

# Calculate the classification accuracy for each model
calcRes = function(fitObj, trueY, mode, test.x, time) {

  # Selected mode depending on supervised model
  if (mode == 1) {
    prob = predict(fitObj, as.matrix(test.x), type = "response")
    pred_rocr = prediction(prob, trueY)
  } else if (mode == 2) {
    prob = predict(fitObj, test.x, type = "prob")[,2]
    pred_rocr = prediction(prob, trueY)
  } else if (mode == 3) {
    prob = predict(fitObj, test.x, type="prob", probability = TRUE)
    pred_rocr = prediction(attr(prob, "probabilities")[,2], trueY)
  } else if (mode == 4) {
    prob = attributes(fitObj)$prob
    pred_rocr = prediction(prob, trueY)
  }

  # Calculate the AUC
  auc = performance(pred_rocr, "auc")@y.values[[1]]
  if (auc < 0.5) { auc = 1 - auc }

  # Calculate the accuracy
  if (mode != 4) {
    pred = predict(fitObj, as.matrix(test.x), type = "class")
  } else {
    pred = fitObj
  }
  accu = mean(pred == trueY)

  # Sensitivity
  sense = sensitivity(factor(pred), factor(trueY))
```

```
  # Specificity
  spec = specificity(factor(pred), factor(trueY))

  return(c(accu, auc, sense, spec, time))
}
```

A.1 Image Preprocessing

```
###################################################
# Read the filenames in a folder #

# This reads all the file names and put them in ascending order
# as images are labeled in numerical order in our dataset
def readData(inDir, mode):

    # Read in the files
    regex = re.compile(r'\d+')
    imgfiles = glob.glob(inDir + '/*.jpg')
    imgfiles = [f for f in glob.glob(inDir + "**/*.jpg", recursive =
True)]

    # Extract image name and sort in ascending order
    if mode == 1:
        imgnum = [0] * len(imgfiles)
        for i in range(0, len(imgfiles)):
            imgnum[i] = [int(x) for x in regex.findall(imgfiles[i])][0]
        filenum = pd.DataFrame(list(zip(imgfiles, imgnum)), columns =
['files','index'])
        filenum = filenum.sort_values('index')
        filenum = filenum.set_index('index')

        return(filenum['files'].to_numpy())

    return imgfiles

################################################################
# Apply median filtering with 3x3 windows and output to file
ogfiles = readData('./og_imgs/', 1)

# Apply median filters to all images in file
for imgf in ogfiles:
    imgname = re.findall(r"[\w']+|[.,!?;]", imgf)[2]
    img = cv2.imread(imgf)
    median = cv2.medianBlur(img, 3)
    cv2.imwrite('./og_med_filter/' + imgname + ".jpg", median)

# Hair Removal function
def dullrazor(imgname, outdir, lowbound=15, showimgs=True,
```

```
filterstruc=3, inpaintmat=3):

    img = cv2.imread(imgname)

    #grayscale
    imgtmp1 = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    #applying a blackhat
    filterSize =(filterstruc, filterstruc)
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, filterSize)
    imgtmp2 = cv2.morphologyEx(imgtmp1, cv2.MORPH_BLACKHAT, kernel)

    #0=skin and 255=hair
    ret, mask = cv2.threshold(imgtmp2, lowbound, 255,
cv2.THRESH_BINARY)

    #inpainting
    img_final = cv2.inpaint(img, mask, inpaintmat ,cv2.INPAINT_TELEA)

    if showimgs:
        print("_____DULLRAZOR_____")
        plt.imshow(imgtmp1, cmap="gray")
        plt.show()
        plt.imshow(imgtmp2, cmap='gray')
        plt.show()
        plt.imshow(mask, cmap='gray')
        plt.show()
        plt.imshow(img_final)
        plt.show()
        print("___________________")

    outname = imgname.split("\\")[1]
    cv2.imwrite("./" + outdir + '/' + outname, img_final)


# Shades of Grey Normalization
def shade_of_gray_cc(imgname, outdir, power=6, gamma=None):
    """
    img (numpy array): the original image with format of (h, w, c)
    power (int): the degree of norm, 6 is used in reference paper
    gamma (float): the value of gamma correction, 2.2 is used in
reference paper
    """
    img = cv2.imread(imgname)

    img_dtype = img.dtype

    if gamma is not None:
        img = img.astype('uint8')
```

```
        look_up_table = np.ones((256,1), dtype='uint8') * 0
        for i in range(256):
            look_up_table[i][0] = 255 * pow(i/255, 1/gamma)
        img = cv2.LUT(img, look_up_table)

    img = img.astype('float32')
    img_power = np.power(img, power)
    rgb_vec = np.power(np.mean(img_power, (0,1)), 1/power)
    rgb_norm = np.sqrt(np.sum(np.power(rgb_vec, 2.0)))
    rgb_vec = rgb_vec/rgb_norm
    rgb_vec = 1/(rgb_vec*np.sqrt(3))
    img = np.multiply(img, rgb_vec)

    # Andrew Anikin suggestion
    img = np.clip(img, a_min=0, a_max=255)

    outname = imgname.split("\\")[1]
    cv2.imwrite("./" + outdir + '/' + outname, img)

    return img.astype(img_dtype)
```

A.2 Lesion Segmentation

A.2.1 Train U-net to Generate Lesion Segmentation Mask

```
### Run the U-Net ###
#########################################

# Read the image names in a folder
def readData(inDir):

    # Read in the files
    regex = re.compile(r'\d+')
    imgfiles = glob.glob(inDir + '/*.jpg')
    imgfiles = [f for f in glob.glob(inDir + "**/*.jpg", recursive =
True)]

    # Extract image name and sort in ascending order
    imgnum = [0] * len(imgfiles)
    for i in range(0, len(imgfiles)):
        imgnum[i] = [int(x) for x in regex.findall(imgfiles[i])][0]
    filenum = pd.DataFrame(list(zip(imgfiles, imgnum)), columns =
['files','index'])
    filenum = filenum.sort_values('index')

    filenum = filenum.set_index('index')

    return(filenum)
```

```
# Get and resize train images and masks
def get_data(path, imgSize):

    filenum = readData(path);

    X = np.zeros((len(filenum), imgSize, imgSize, 1), dtype=np.float32)

    for n in range(len(filenum)):
        # Load images
        img = load_img(filenum.iloc[n]['files'] , grayscale=True)
        x_img = img_to_array(img)
        x_img = resize(x_img, (imgSize, imgSize, 1), mode='constant',
preserve_range=True)
        # Save images
        X[n, ..., 0] = x_img.squeeze() / 255

    # Force binary if mask
    if (path.find("mask") != -1):
        X[X>0.5]= 1
        X[X<0.5] = 0

    return X

####################################################################
#==================================================================#

# Define the unet model
from tensorflow.python.keras.models import Model, load_model
from tensorflow.python.keras.layers import Input, BatchNormalization,
Activation, Dense, Dropout
from tensorflow.python.keras.layers.core import Lambda, RepeatVector,
Reshape
from tensorflow.python.keras.layers.convolutional import Conv2D,
Conv2DTranspose
from tensorflow.python.keras.layers.pooling import MaxPooling2D,
GlobalMaxPool2D
from tensorflow.python.keras.layers.merge import concatenate, add
from tensorflow.python.keras.callbacks import EarlyStopping,
ModelCheckpoint, ReduceLROnPlateau
from tensorflow.python.keras.optimizers import Adam
from tensorflow.python.keras.preprocessing.image import
ImageDataGenerator, array_to_img, img_to_array, load_img

# Define a 2d convolution block
def conv2d_block(input_tensor, n_filters, kernel_size=3,
batchnorm=True):
    # first layer
    x = Conv2D(filters=n_filters, kernel_size=(kernel_size,
```

```
kernel_size),
    kernel_initializer="he_normal",
               padding="same")(input_tensor)
    if batchnorm:
        x = BatchNormalization()(x)
    x = Activation("relu")(x)
    # second layer
    x = Conv2D(filters=n_filters, kernel_size=(kernel_size,
kernel_size),
    kernel_initializer="he_normal",
               padding="same")(x)
    if batchnorm:
        x = BatchNormalization()(x)
    x = Activation("relu")(x)
    return x

# Construct a Unet model
def get_unet(input_img, n_filters=16, dropout=0.5, batchnorm=True):
    # contracting path
    c1 = conv2d_block(input_img, n_filters=n_filters*1, kernel_size=3,
batchnorm=batchnorm)
    p1 = MaxPooling2D((2, 2)) (c1)
    p1 = Dropout(dropout*0.5)(p1)

    c2 = conv2d_block(p1, n_filters=n_filters*2, kernel_size=3,
batchnorm=batchnorm)
    p2 = MaxPooling2D((2, 2)) (c2)
    p2 = Dropout(dropout)(p2)

    c3 = conv2d_block(p2, n_filters=n_filters*4, kernel_size=3,
batchnorm=batchnorm)
    p3 = MaxPooling2D((2, 2)) (c3)
    p3 = Dropout(dropout)(p3)

    c4 = conv2d_block(p3, n_filters=n_filters*8, kernel_size=3,
batchnorm=batchnorm)
    p4 = MaxPooling2D(pool_size=(2, 2)) (c4)
    p4 = Dropout(dropout)(p4)

    c5 = conv2d_block(p4, n_filters=n_filters*16, kernel_size=3,
batchnorm=batchnorm)

    # expansive path
    u6 = Conv2DTranspose(n_filters*8, (3, 3), strides=(2, 2),
padding='same') (c5)
    u6 = concatenate([u6, c4])
    u6 = Dropout(dropout)(u6)
    c6 = conv2d_block(u6, n_filters=n_filters*8, kernel_size=3,
batchnorm=batchnorm)
```

```
    u7 = Conv2DTranspose(n_filters*4, (3, 3), strides=(2, 2),
padding='same') (c6)
    u7 = concatenate([u7, c3])
    u7 = Dropout(dropout)(u7)
    c7 = conv2d_block(u7, n_filters=n_filters*4, kernel_size=3,
batchnorm=batchnorm)

    u8 = Conv2DTranspose(n_filters*2, (3, 3), strides=(2, 2),
padding='same') (c7)
    u8 = concatenate([u8, c2])
    u8 = Dropout(dropout)(u8)
    c8 = conv2d_block(u8, n_filters=n_filters*2, kernel_size=3,
batchnorm=batchnorm)

    u9 = Conv2DTranspose(n_filters*1, (3, 3), strides=(2, 2),
padding='same') (c8)
    u9 = concatenate([u9, c1], axis=3)
    u9 = Dropout(dropout)(u9)
    c9 = conv2d_block(u9, n_filters=n_filters*1, kernel_size=3,
batchnorm=batchnorm)

    outputs = Conv2D(1, (1, 1), activation='sigmoid') (c9)
    model = Model(inputs=[input_img], outputs=[outputs])
    return model

# Set up the model
imgSize = 256
input_img = Input((imgSize, imgSize, 1), name='img')
model = get_unet(input_img, n_filters=16, dropout=0.05, batchnorm=True)
model.compile(optimizer="adam", loss="binary_crossentropy",
metrics=["acc"])
model.summary()

#######################################################################
#########
### Train the model ###

# Set up the data tensors
# Set up directory for training, mask, and target testing set
trainDir = './image/'
trainMaDir= './mask/'

testDir = "./temp2/"
testMaDir = "./utemp/"

# Use get_data instead
trainImgs = get_data(trainDir, 256)
trainMask = get_data(trainMaDir, 256)
```

```
testImgs = get_data(testDir, 256)
testMask = get_data(testMaDir, 256)

# Image data generator distortion options
data_gen_args = dict(rotation_range=45.,
                     width_shift_range=0.1,
                     height_shift_range=0.1,
                     shear_range=0.2,
                     zoom_range=0.2,
                     horizontal_flip=True,
                     vertical_flip=True,
                     fill_mode='reflect')


imgtrain = train_datagen(trainImgs)


model.fit(trainImgs, trainMask, batch_size = 30, epochs = 1000)
res = model.predict(trainImgs)

# Run test
runTrain = model.predict(trainImgs)
runTest = model.predict(testImgs)
binTest = binImage(runTest)

# Output the images
for i in range(len(binTest)):
    save_img("./utemp/" + str(i) + ".jpg", binTest[i])
```

A.2.2 Post-processing of Generated Lesion Masks

```
### Read in the data ###
def readData(inDir):

    # Read in the files
    regex = re.compile(r'\d+')
    imgfiles = glob.glob(inDir + '/*.jpg')
    imgfiles = [f for f in glob.glob(inDir + "**/*.jpg", recursive =
True)]

    # Extract image name and sort in ascending order
    imgnum = [0] * len(imgfiles)
    for i in range(0, len(imgfiles)):
        imgnum[i] = [int(x) for x in regex.findall(imgfiles[i])][0]
    filenum = pd.DataFrame(list(zip(imgfiles, imgnum)), columns =
['files','index'])
    filenum = filenum.sort_values('index')

    filenum = filenum.set_index('index')
```

```python
    return(filenum)

### Get and resize train images and masks ###
def get_data(path, imgSize, gray, mask):

    filenum = readData(path);
    if gray:
        chan = 1
    else:
        chan = 3

    X = np.zeros((len(filenum), imgSize, imgSize, chan),
dtype=np.float32)

    for n in range(len(filenum)):
        # Load images
        img = load_img(filenum.iloc[n]['files'], grayscale=gray)
        x_img = img_to_array(img)
        x_img = resize(x_img, (imgSize, imgSize, chan),
mode='constant', preserve_range=True)

        # Save images
        X[n] = x_img

    # Force binary if mask
    if mask:
        X[X>0.5]= 1
        X[X<0.5] = 0

    return X

### Binarize an image ###
def binImage(imageNP):
    imageNP[imageNP > 0.5] = 1
    imageNP[imageNP < 0.5] = 0
    return imageNP


#######################################################################
########
### Post-processing of segmentation mask ###

'''
1. Retain only the largest blob
2. Fill in the small holes
'''
# Wrapper function to post-process the segmentation masks
def postProc(imageNP, fileName):
```

```
    # Binarize image array
    imageNP = binImage(imageNP)

    for i in range(len(imageNP)):
        dumImg = removeSpots(imageNP[i])
        dumImg = dumImg[:,:,None]
        save_img(fileName + str(i+1) +".jpg", dumImg)

# Remove holes and spots on the mask
def removeSpots(img1):

    # Gaussian smoothing on the masks
    img1 = cv2.GaussianBlur(img1, (3,3), 0)
    plt.imshow(img1.squeeze(), cmap = "gray")

    # Largest blob of original image
    blobImg = largeBlob(img1)
    plt.imshow(blobImg.squeeze(), cmap = "gray")

    # fill in the remaining holes
    cleanImg = ndimage.binary_fill_holes(blobImg).astype(int)
    plt.imshow(cleanImg.squeeze(), cmap = "gray")
    cleanImg = largeBlob(cleanImg)

    return(cleanImg)

# find the largest color blob in image
def largeBlob(img1):

    # Generate intermediate image; use morphological closing to keep
parts of the brain together
    img1 = np.uint8(img1 * 255)

    # Find largest contour in intermediate image
    cnts, _ = cv2.findContours(img1, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_NONE)
    cnt = max(cnts, key=cv2.contourArea)

    # Output
    out = np.zeros(img1.shape, np.uint8)
    cv2.drawContours(out, [cnt], -1, 255, cv2.FILLED)
    out = cv2.bitwise_and(img1, out)

    return out

# Read generated mask from "in_mask" folder
# Output post-processed mask into "out_mask" folder
```

```
testmask = get_data("./in_mask/", 256, True, True)
postProc(testmask, "./out_mask/")
```