Toward automated severe pharyngitis detection with smartphone camera using deep learning networks

Tae Keun Yoo,[a,*] Joon Yul Choi,[b,*] Yeon Il Jang,[c] Ein Oh[d]

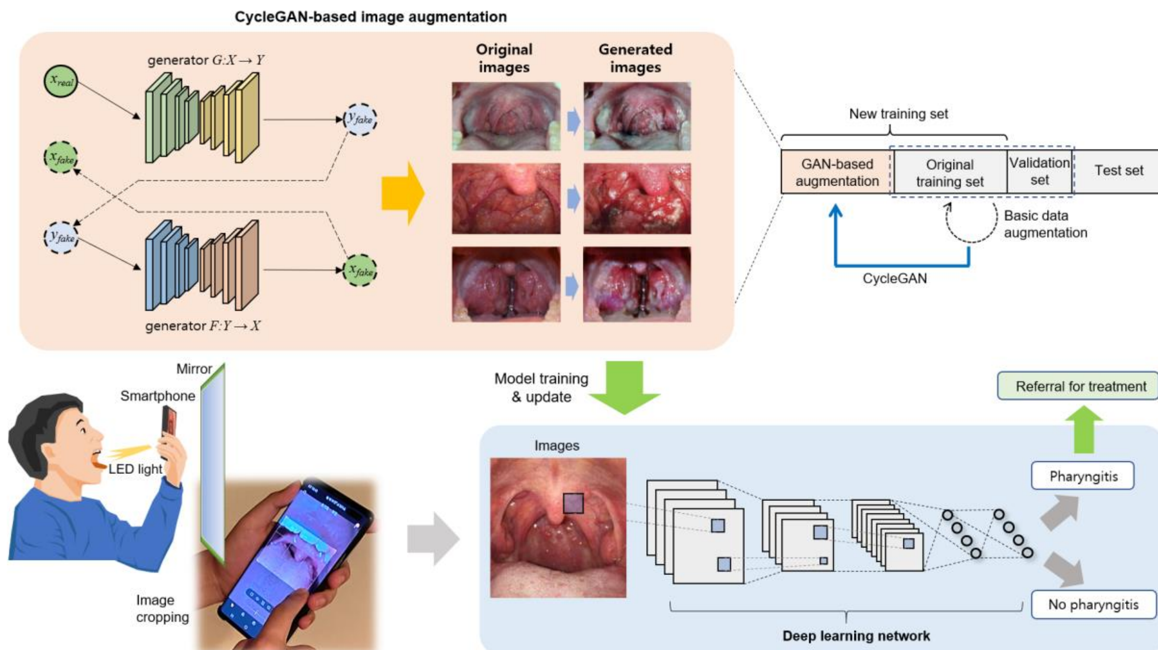
[a] Department of Ophthalmology, Aerospace Medical Center, Republic of Korea Air Force, Cheongju, South Korea

[b] Epilepsy Center, Neurological Institute, Cleveland Clinic, Cleveland, Ohio

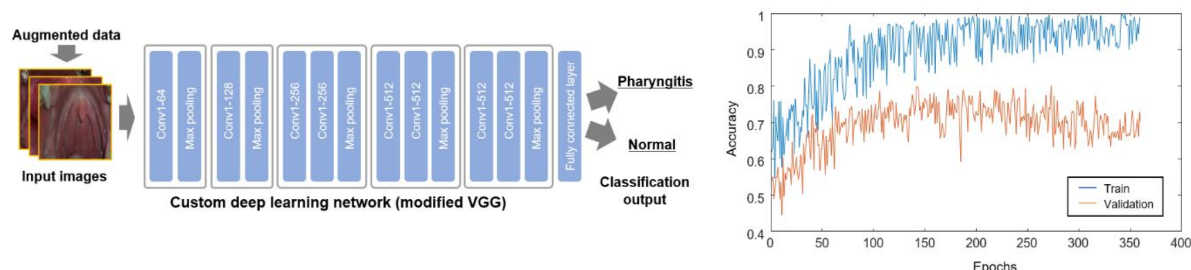
[c] Department of Otorhinolaryngology-Head & Neck Surgery, 10th Fighter Wing, Republic of Korea Air Force, Cheongju, South Korea

[d] Department of Anesthesiology and Pain Medicine, Seoul Women's Hospital, Bucheon, South Korea

Graphical abstract



Supplementary Materials. Custom deep learning network for pharyngitis detection



Every convolutional layer block was followed by batch normalizations for the regularization of the model. Adam optimizer with fixed weight decay was used for training our model. The learning rate was set to 1e-4, batch size of 32 was used and training was performed during 360 epochs.

When we trained a custom deep learning network without the transfer learning technique, the validation accuracy was lower than 80% and was not improved during the training epochs. In fact, custom convolutional neural network requires back and forth fine-tuning of the model layers and huge volumes of annotated data for improved accuracy which may not be available for medical image analysis.

Supplementary Materials. Codes for Grad-CAM

The original code being used in the current study for developing the Grad-CAM algorithm is based on

https://colab.research.google.com/drive/1xVA1kJXOjPPjp32uRKHM5qJTSpDP7Qj1.

Supplementary Materials. Codes for convolutional neural networks (Inception-v3) and generative adversarial network (CycleGAN)

Codes for Inception-v3

```
# based on the web site:
https://colab.research.google.com/github/google/eng-
edu/blob/master/ml/pc/exercises/image_classification_part3.ipynb#sc
rollTo=zE37ARlqY9da
# connect data in your google drive
from google.colab import auth
auth.authenticate_user()

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

train_dir ='./gdrive/My Drive/SoreThroat/Train'
val_dir = './gdrive/My Drive/ SoreThroat /Val'

# Note that the validation data should not be augmented!
val_datagen = ImageDataGenerator(rescale=1./255)
```

```
train_generator = train_datagen.flow_from_directory(
        train_dir,   # This is the source directory for training images
        target_size=(256, 256),   # All images will be resized to
256x256
        batch_size=20,
        # Since we use binary_crossentropy loss, we need binary
labels
        class_mode='categorical')

validation_generator =   val_datagen.flow_from_directory(
        val_dir,   # This is the source directory for training images
        target_size=(256, 256),   # All images will be resized to
256x256
        batch_size=20,
        shuffle=False,
        # Since we use binary_crossentropy loss, we need binary
labels
        class_mode='categorical')
```

```
import os

from tensorflow.keras import layers
from tensorflow.keras import Model

!wget --no-check-certificate \
    https://storage.googleapis.com/mledu-datasets/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5 \
    -O /tmp/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5

from tensorflow.keras.applications.inception_v3 import InceptionV3

local_weights_file = '/tmp/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5'
pre_trained_model = InceptionV3(
    input_shape=(256, 256, 3), include_top=False, weights=None)
pre_trained_model.load_weights(local_weights_file)

for layer in pre_trained_model.layers:
  layer.trainable = False

last_layer = pre_trained_model.get_layer('mixed7')
print('last layer output shape:', last_layer.output_shape)
last_output = last_layer.output


# Flatten the output layer to 1 dimension
x = layers.Flatten()(last_output)
# Add a fully connected layer with 1,024 hidden units and ReLU activation
x = layers.Dense(1024, activation='relu')(x)
# Add a dropout rate of 0.2
x = layers.Dropout(0.2)(x)
# Add a final sigmoid layer for classification
x = layers.Dense(2, activation='sigmoid')(x)
### Important : 2 = Number of classes

# Configure and compile the model
model = Model(pre_trained_model.input, x)

unfreeze = False

# Unfreeze all models after "mixed6"
for layer in pre_trained_model.layers:
  if unfreeze:
    layer.trainable = True
  if layer.name == 'mixed6':
    unfreeze = True

model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['acc'])

history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=1000,
      validation_data=validation_generator,
      validation_steps=50,
      verbose=2)

acc = history.history['acc']
val_acc = history.history['val_acc']

loss = history.history['loss']
val_loss = history.history['val_loss']
```

Codes for CycleGAN

```
# based on the web site
https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/cyclegan.ipynb#scrollTo=xE-4QFEjSpEu


!pip install git+https://github.com/tensorflow/examples.git

try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass
import tensorflow as tf

from __future__ import absolute_import, division, print_function, unicode_literals

import tensorflow_datasets as tfds
from tensorflow_examples.models.pix2pix import pix2pix

import os
import time
import matplotlib.pyplot as plt
from IPython.display import clear_output

tfds.disable_progress_bar()
AUTOTUNE = tf.data.experimental.AUTOTUNE


# connect data in your google drive
from google.colab import auth
auth.authenticate_user()

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

import os
from glob import glob

from PIL import Image
import numpy as np

#input pipeline

BUFFER_SIZE = 2000
BATCH_SIZE = 1
IMG_WIDTH = 256
IMG_HEIGHT = 256


def load(image_file):
  image = tf.io.read_file(image_file)
  image = tf.image.decode_jpeg(image)

  input_image = image[:, :, :]

  input_image = tf.cast(input_image, tf.float32)

  return input_image


def resize(input_image, height, width):
  input_image = tf.image.resize(input_image, [height, width],
method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)

  return input_image


def normalize(input_image):
  input_image = (input_image / 127.5) - 1

  return input_image


@tf.function()
def random_jitter(input_image):
  # resizing to 265 x 265 x 3
  input_image = resize(input_image, 265, 265)

  # randomly cropping to 256 x 256 x 3
  input_image = random_crop(input_image)

  if tf.random.uniform(()) > 0.5:
    # random mirroring
    input_image = tf.image.flip_left_right(input_image)

  return input_image

def load_image_train(image_file):
  input_image = load(image_file)
  input_image = random_jitter(input_image)
  input_image = normalize(input_image)

  return input_image
```

```
def load_image_test(image_file):
   input_image = load(image_file)
   input_image = resize(input_image,
                                                         IMG_HEIGHT, IMG_WIDTH)
   input_image = normalize(input_image)

   return input_image

def random_crop(input_image):
   stacked_image = tf.stack([input_image, input_image], axis=0)
   cropped_image = tf.image.random_crop(
         stacked_image, size=[2, IMG_HEIGHT, IMG_WIDTH, 3])

   return cropped_image[0]



train_datasetA = tf.data.Dataset.list_files('./gdrive/My
Drive/Data/trainA/*.png')
train_datasetA = train_datasetA.map(load_image_train,
num_parallel_calls=AUTOTUNE)
train_datasetA = train_datasetA.shuffle(BUFFER_SIZE)
train_datasetA = train_datasetA.batch(BATCH_SIZE)

train_datasetB = tf.data.Dataset.list_files('./gdrive/My
Drive/Data/trainB/*.png')
train_datasetB = train_datasetB.map(load_image_train,
num_parallel_calls=AUTOTUNE)
train_datasetB = train_datasetB.shuffle(BUFFER_SIZE)
train_datasetB = train_datasetB.batch(BATCH_SIZE)

test_datasetA = tf.data.Dataset.list_files('./gdrive/My
Drive/Data/testA/*.png')
test_datasetA = test_datasetA.map(load_image_test,
num_parallel_calls=AUTOTUNE)
test_datasetA = test_datasetA.shuffle(BUFFER_SIZE)
test_datasetA = test_datasetA.batch(BATCH_SIZE)


test_datasetB = tf.data.Dataset.list_files('./gdrive/My
Drive/Data/testB/*.png')
test_datasetB = test_datasetB.map(load_image_test,
num_parallel_calls=AUTOTUNE)
test_datasetB = test_datasetB.shuffle(BUFFER_SIZE)
test_datasetB = test_datasetB.batch(BATCH_SIZE)

# standard cyclegan을 써서 horse zebra 용어는 그대로 두겠다.
sample_horse = next(iter(train_datasetA))
sample_zebra = next(iter(train_datasetB))

OUTPUT_CHANNELS = 3

generator_g = pix2pix.unet_generator(OUTPUT_CHANNELS,
norm_type='instancenorm')
generator_f = pix2pix.unet_generator(OUTPUT_CHANNELS,
norm_type='instancenorm')

discriminator_x = pix2pix.discriminator(norm_type='instancenorm',
target=False)
discriminator_y = pix2pix.discriminator(norm_type='instancenorm',
target=False)


to_zebra = generator_g(sample_horse)
to_horse = generator_f(sample_zebra)
plt.figure(figsize=(8, 8))
contrast = 8

imgs = [sample_horse, to_zebra, sample_zebra, to_horse]
title = ['Horse', 'To Zebra', 'Zebra', 'To Horse']

for i in range(len(imgs)):
   plt.subplot(2, 2, i+1)
   plt.title(title[i])
   if i % 2 == 0:
      plt.imshow(imgs[i][0] * 0.5 + 0.5)
   else:
      plt.imshow(imgs[i][0] * 0.5 * contrast + 0.5)
plt.show()


plt.figure(figsize=(8, 8))

plt.subplot(121)
plt.title('Is a real zebra?')
plt.imshow(discriminator_y(sample_zebra)[0, ..., -1], cmap='RdBu_r')

plt.subplot(122)
plt.title('Is a real horse?')
plt.imshow(discriminator_x(sample_horse)[0, ..., -1], cmap='RdBu_r')

plt.show()
LAMBDA = 10
loss_obj = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real, generated):
   real_loss = loss_obj(tf.ones_like(real), real)

   generated_loss = loss_obj(tf.zeros_like(generated), generated)

   total_disc_loss = real_loss + generated_loss

   return total_disc_loss * 0.5


def generator_loss(generated):
   return loss_obj(tf.ones_like(generated), generated)



def calc_cycle_loss(real_image, cycled_image):
   loss1 = tf.reduce_mean(tf.abs(real_image - cycled_image))

   return LAMBDA * loss1


def identity_loss(real_image, same_image):
   loss = tf.reduce_mean(tf.abs(real_image - same_image))
   return LAMBDA * 0.5 * loss


generator_g_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
generator_f_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)

discriminator_x_optimizer = tf.keras.optimizers.Adam(2e-4,
beta_1=0.5)
discriminator_y_optimizer = tf.keras.optimizers.Adam(2e-4,
beta_1=0.5)


checkpoint_path = "./checkpoints/train"

ckpt = tf.train.Checkpoint(generator_g=generator_g,
                                             generator_f=generator_f,
                                             discriminator_x=discriminator_x,
                                             discriminator_y=discriminator_y,

generator_g_optimizer=generator_g_optimizer,

generator_f_optimizer=generator_f_optimizer,

discriminator_x_optimizer=discriminator_x_optimizer,

discriminator_y_optimizer=discriminator_y_optimizer)

ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path,
max_to_keep=5)

# if a checkpoint exists, restore the latest checkpoint.
if ckpt_manager.latest_checkpoint:
   ckpt.restore(ckpt_manager.latest_checkpoint)
   print ('Latest checkpoint restored!!')



EPOCHS = 100

def generate_images(model, test_input):
   prediction = model(test_input)

   plt.figure(figsize=(12, 12))

   display_list = [test_input[0], prediction[0]]
   title = ['Input Image', 'Predicted Image']

   for i in range(2):
      plt.subplot(1, 2, i+1)
```

```
        plt.title(title[i])
        # getting the pixel values between [0, 1] to plot it.
        plt.imshow(display_list[i] * 0.5 + 0.5)
        plt.axis('off')
    plt.show()


@tf.function
def train_step(real_x, real_y):
    # persistent is set to True because the tape is used more than
    # once to calculate the gradients.
    with tf.GradientTape(persistent=True) as tape:
        # Generator G translates X -> Y
        # Generator F translates Y -> X.

        fake_y = generator_g(real_x, training=True)
        cycled_x = generator_f(fake_y, training=True)

        fake_x = generator_f(real_y, training=True)
        cycled_y = generator_g(fake_x, training=True)

        # same_x and same_y are used for identity loss.
        same_x = generator_f(real_x, training=True)
        same_y = generator_g(real_y, training=True)

        disc_real_x = discriminator_x(real_x, training=True)
        disc_real_y = discriminator_y(real_y, training=True)

        disc_fake_x = discriminator_x(fake_x, training=True)
        disc_fake_y = discriminator_y(fake_y, training=True)

        # calculate the loss
        gen_g_loss = generator_loss(disc_fake_y)
        gen_f_loss = generator_loss(disc_fake_x)

        total_cycle_loss = calc_cycle_loss(real_x, cycled_x) +
calc_cycle_loss(real_y, cycled_y)

        # Total generator loss = adversarial loss + cycle loss
        total_gen_g_loss = gen_g_loss + total_cycle_loss +
identity_loss(real_y, same_y)
        total_gen_f_loss = gen_f_loss + total_cycle_loss +
identity_loss(real_x, same_x)

        disc_x_loss = discriminator_loss(disc_real_x, disc_fake_x)
        disc_y_loss = discriminator_loss(disc_real_y, disc_fake_y)

    # Calculate the gradients for generator and discriminator
    generator_g_gradients = tape.gradient(total_gen_g_loss,

generator_g.trainable_variables)
    generator_f_gradients = tape.gradient(total_gen_f_loss,

generator_f.trainable_variables)

    discriminator_x_gradients = tape.gradient(disc_x_loss,

discriminator_x.trainable_variables)
    discriminator_y_gradients = tape.gradient(disc_y_loss,

discriminator_y.trainable_variables)

    # Apply the gradients to the optimizer
    generator_g_optimizer.apply_gradients(zip(generator_g_gradients,

generator_g.trainable_variables))

    generator_f_optimizer.apply_gradients(zip(generator_f_gradients,

generator_f.trainable_variables))


discriminator_x_optimizer.apply_gradients(zip(discriminator_x_gradie
nts,

discriminator_x.trainable_variables))


discriminator_y_optimizer.apply_gradients(zip(discriminator_y_gradie
nts,

discriminator_y.trainable_variables))


for epoch in range(EPOCHS):
    start = time.time()

    n = 0
    for image_x, image_y in tf.data.Dataset.zip((train_datasetA,
train_datasetB)):
        train_step(image_x, image_y)
        if n % 10 == 0:
            print ('.', end='')
        n+=1

    clear_output(wait=True)
    # Using a consistent image (sample_horse) so that the progress of
the model
    # is clearly visible.
    generate_images(generator_g, sample_horse)

    if (epoch + 1) % 5 == 0:
        ckpt_save_path = ckpt_manager.save()
        print ('Saving checkpoint for epoch {} at {}'.format(epoch+1,

ckpt_save_path))

    print ('Time taken for epoch {} is {} sec\n'.format(epoch + 1,

time.time()-start))

# Run the trained model on the test dataset
for inp in test_datasetA.take(50):
    generate_images(generator_g, inp)
```