UDACITY MACHINE LEARNING NANODEGREE 2019

CAPSTONE PROJECT

Detecting Malaria Using Deep Learning CNN

Prabhat Sharma

21th June 2019

DEFINITION

Project Overview

Malaria is a deadly, infectious, mosquito-borne disease caused by Plasmodium parasites that are transmitted by the bite of infected female Anopheles mosquitoes. If an infected mosquito bites you, parasites carried by the mosquito enter your blood and start destroying oxygen-carrying red blood cells (RBC). Typically, the first symptoms of malaria are similar to a virus like the flu and they usually begin within a few days or weeks after the mosquito bite. However, these deadly parasites can live in your body for over a year without causing symptoms, and a delay in treatment can lead to complications and even death. Therefore, early detection can save lives. The World Health Organization's (WHO) malaria facts indicate that nearly half the world's population is at risk from malaria, and there are over 200 million malaria cases and approximately 400,000 deaths due to malaria every year. This is a motivation to make malaria detection and diagnosis fast, easy, and effective.

Detection¶

According to WHO protocol, diagnosis typically involves intensive examination of the blood smear at 100X magnification. Trained people manually count how many red blood cells contain parasites out of 5,000 cells. As the Rajaraman, et al., paper cited above explains :

Thick blood smears assist in detecting the presence of parasites while thin blood smears assist in identifying the species of the parasite causing the infection (Centers for Disease Control and Prevention, 2012). The diagnostic accuracy heavily depends on human expertise and can be adversely impacted by the inter-observer variability and the liability imposed by large-scale diagnoses in disease-endemic/resource-constrained regions (Mitiku, Mengistu, and Gelaw, 2003). Alternative techniques such as polymerase chain reaction (PCR) and rapid diagnostic tests (RDT) are used; however, PCR analysis is limited in its performance (Hommelsheim, et al., 2014) and RDTs are less cost-effective in disease-endemic regions (Hawkes, Katsuva, and Masumbuko, 2009).

Thus, malaria detection could benefit from automation using deep learning.

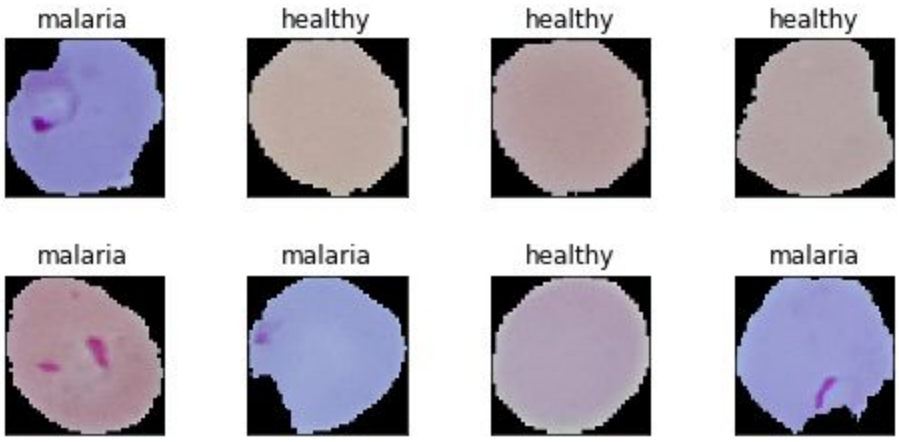
Deep learning for malaria detection¶

Deep learning models, or more specifically convolutional neural networks (CNNs), have proven very effective in a wide variety of computer vision tasks. Convolution layers learn spatial hierarchical patterns from data, which are also translation-invariant, so they are able to learn different aspects of images. This allows CNNs to automate feature engineering and learn effective features that generalize well on new data points. Pooling layers helps with downsampling and dimension reduction.

Our focus is to try some simple CNN models from scratch and a couple of pre-trained models using transfer learning.

Problem Statement

The goal is to create a deep learning Convolution Neural Network model which can predict if the image of cell is malaria infected or uninfected.



Above shown is a grid of malaria infected cells and healthy cells. There is a visible difference in them which is partially detectable by human eye. We need to train a CNN model which can also find out and differentiate between these two types of cell. Thus we will consider this as an image classification problem. To achieve this, the plan is to implement CNN and transfer learning abilities of CNNs.

Metrics

The evaluation metric will be as follows :-

Accuracy is a common metric for binary classifiers; it takes into account both true positives and true negatives with equal weight.

$$Accuracy = (True\ Positives + True\ Negatives)/Dataset\ Size$$

F1 score is the harmonic mean between precision and recall. F1 score provide the measure of precision and robustness of model.

$$F1 = 2 * (precision * recall)/(precision + recall)$$

Recall is the ability of the model to find all the relevant cases within the dataset.

$$Recall = True\ Positives/(True\ Positives + False\ Negatives)$$

Performance of a model will be evaluated using F1 score and accuracy. Model need to have high recall value as well, false-negative can not be tolerated for the requirement of malaria detection. False negative result can risk the life of patient. So even though accuracy and precision will also be calculated with recall and F1 score, Recall is the evaluation metric which will be key point of consideration. Minimum of 80% accuracy, precision, recall and F1 score is desired for the best model.

ANALYSIS

Data Exploration

The data for our analysis comes from researchers at the Lister Hill National Center for Biomedical Communications (LHNCBC), part of the National Library of Medicine (NLM), who have carefully collected and annotated the publicly available dataset of healthy and infected blood smear images. Let's briefly describe the dataset's structure. We have two folders that contain images of cells, infected and healthy. Dataset is balanced in nature as there are 13779 images each for infected and uninfected cells.

 # Code

```
import os
import glob

base_dir = os.path.join('./cell_images')
infected_dir = os.path.join(base_dir,'Parasitized')
healthy_dir = os.path.join(base_dir,'Uninfected')

infected_files = glob.glob(infected_dir+'/*.png')
healthy_files = glob.glob(healthy_dir+'/*.png')
len(infected_files), len(healthy_files)
```

#Output

(13779, 13779)

Let's build a data frame from this, which we will use when we start building our datasets.

#Code

```
import numpy as np
import pandas as pd

np.random.seed(42)

files_df = pd.DataFrame({
    'filename': infected_files + healthy_files,
    'label': ['malaria'] * len(infected_files) + ['healthy'] * len(healthy_files)
}).sample(frac=1, random_state=42).reset_index(drop=True)

files_df.head()
```

#Output

	filename	label
0	./cell_images/Parasitized/C59P20thinF_IMG_2015...	malaria
1	./cell_images/Parasitized/C180P141NThinF_IMG_2...	malaria
2	./cell_images/Uninfected/C154P115ThinF_IMG_201...	healthy
3	./cell_images/Uninfected/C69P30N_ThinF_IMG_201...	healthy
4	./cell_images/Uninfected/C182P143NThinF_IMG_20...	healthy

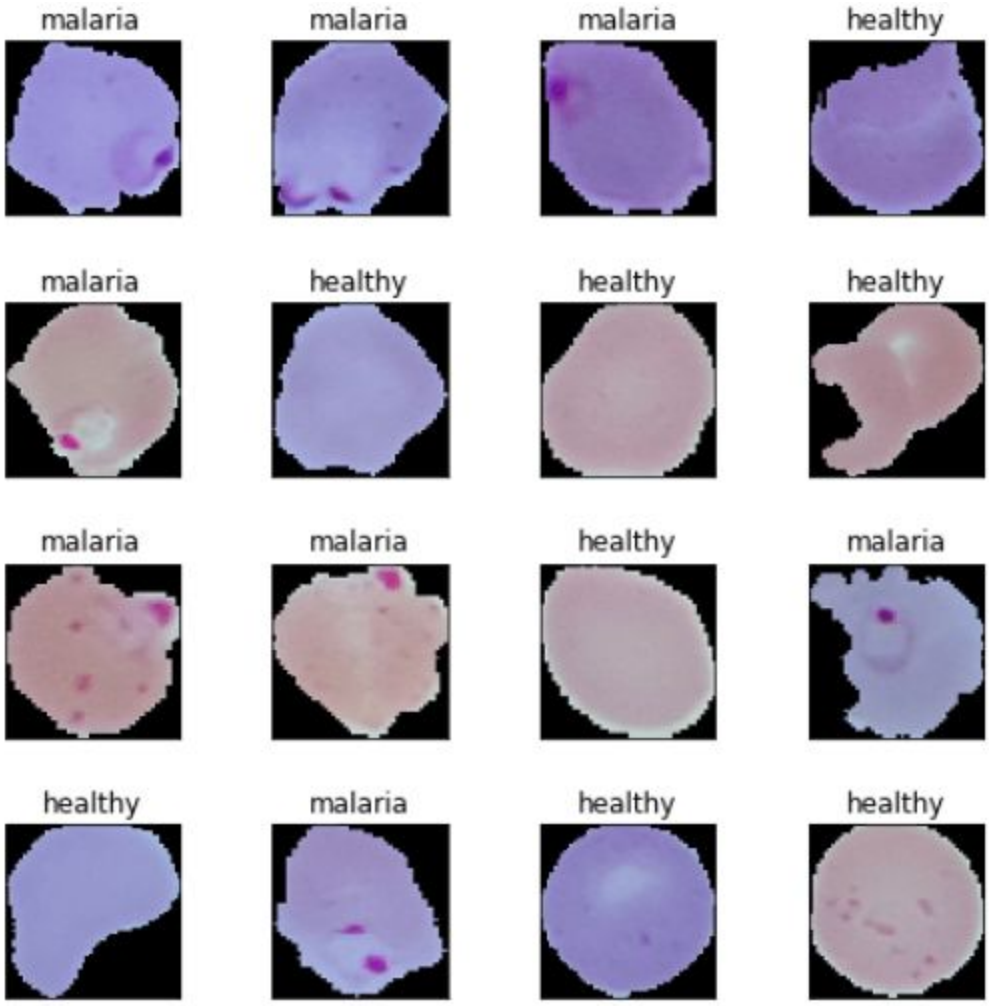
Exploratory Visualization

We can now view some sample cell images to get an idea of how our data looks.

#Code

```
import matplotlib.pyplot as plt

%matplotlib inline

plt.figure(1 , figsize = (8 , 8))

n = 0

for i in range(16):

    n += 1

    r = np.random.randint(0 , files_df.shape[0] , 1)

    plt.subplot(4 , 4 , n)

    plt.subplots_adjust(hspace = 0.5 , wspace = 0.5)

    plt.imshow(files_df[r[0]]/255.)

    plt.title('{}'.format(train_labels[r[0]]))

    plt.xticks([]) , plt.yticks([])
```

#Output



Based on these sample images, we can see some subtle differences between malaria and healthy cell images. We will make our deep learning models try to learn these patterns during model training.

Algorithms and Techniques

In the model training phase, we will build three deep learning models, train them with our training data, and compare their performance using the validation data. We will then save these models and use them later in the model evaluation phase.

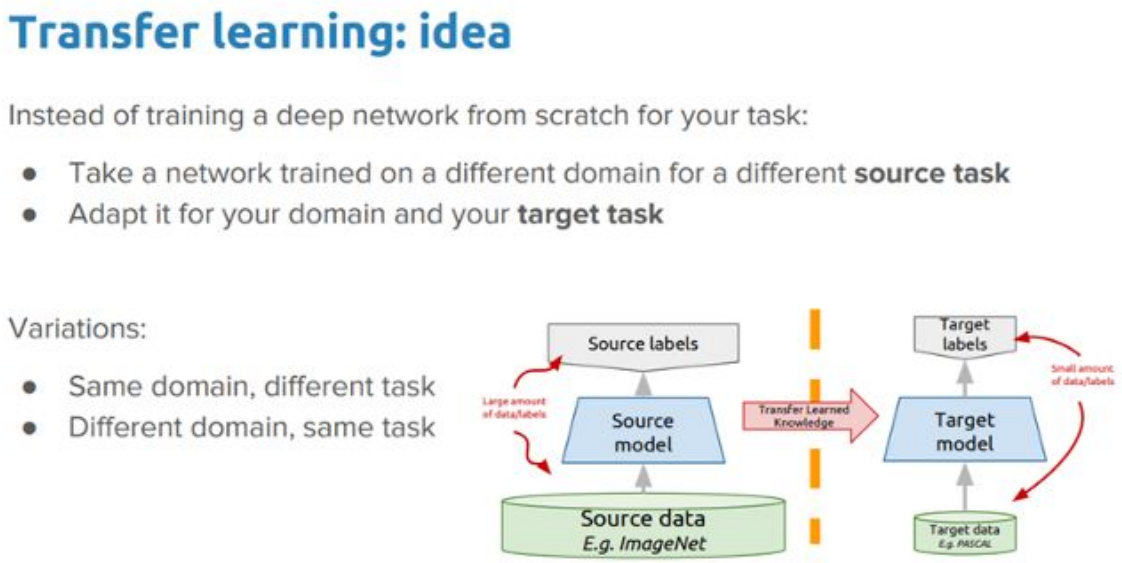
Model 1 : CNN from scatch

Architecture :-

- Three convolution layer(kernel size = (3,3), padding = 'same', activation = 'relu')
- Three pooling layer(pool size = (3,3))
- Two dense layer(nodes = 512, activation = 'relu')
- Dropout for regularization(rate = 0.3)
- Optimizer = 'adam'
- Loss = 'binary cross entropy'
- Metric = accuracy

Deep transfer learning

Just like humans have an inherent capability to transfer knowledge across tasks, transfer learning enables us to utilize knowledge from previously learned tasks and apply it to newer, related ones, even in the context of machine learning or deep learning.



The idea we want to explore in this exercise is: "Can we leverage a pre-trained deep learning model (which was trained on a large dataset, like ImageNet) to solve the problem of malaria detection by applying and transferring its knowledge in the context of our problem?"

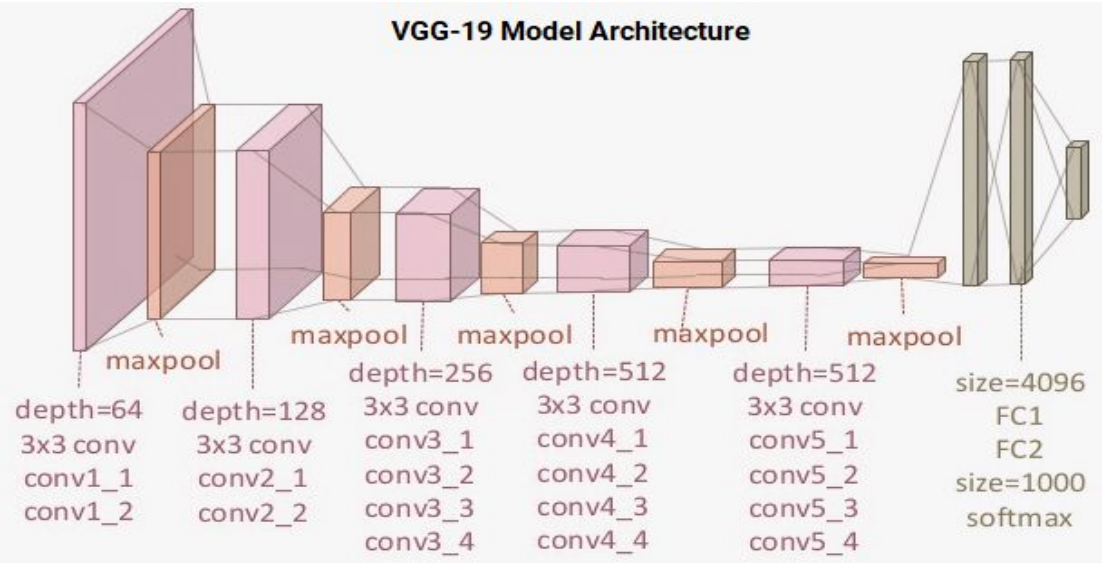
We will apply the two most popular strategies for deep transfer learning.

- Pre-trained model as a feature extractor
- Pre-trained model with fine-tuning

We will be using the pre-trained VGG-19 deep learning models, developed by the Visual Geometry Group (VGG) at the University of Oxford, for our experiments. A pre-trained model like VGG-19 is trained on a huge dataset (ImageNet) with a lot of diverse image categories. Therefore, the model should have learned a robust hierarchy of features, which are spatial-, rotational-, and translation-invariant with regard to features learned by CNN models. Hence, the model, having learned a good representation of features for over a million images, can act as a good feature extractor for new images suitable for computer vision problems like malaria detection.

Understanding the VGG-19 modelUnderstanding the VGG-19 model

The VGG-19 model is a 19-layer (convolution and fully connected) deep learning network built on the ImageNet database, which was developed for the purpose of image recognition and classification. This model was built by Karen Simonyan and Andrew Zisserman and is described in their paper "Very deep convolutional networks for large-scale image recognition." The architecture of the VGG-19 model is:



We have a total of 16 convolution layers using 3x3 convolution filters along with max pooling layers for downsampling and two fully connected hidden layers of 4,096 units in each layer followed by a dense layer of 1,000 units, where each unit represents one of the image categories in the ImageNet database. We do not need the last three layers since we will be using our own fully connected dense layers to predict malaria. We are more concerned with the first five blocks so we can leverage the VGG model as an effective feature extractor.

We will use one of the models as a simple feature extractor by freezing the five convolution blocks to make sure their weights aren't updated after each epoch. For the last model, we will apply fine-tuning to the VGG model, where we will unfreeze the last two blocks (Block 4 and Block 5) so that their weights will be updated in each epoch (per batch of data) as we train our own model.

Model 2 : Pre-trained model as feature extractor

Architecture :-

- Transfer learning model = VGG19(trainable = false)
- Two dense layer(nodes = 512, activation = ‘relu’)
- Optimizer = ‘adam’
- Loss = ‘binary cross entropy’
- Metric = accuracy

Frozen layers of VGG19 will be used as image feature extractor and dense layer will be plugged at the end to perform classification task.

Model 3: Fine-tuned pre-trained model with image augmentation

Architecture :-

- Transfer learning model = VGG19(trainable = true)
- Two dense layer(nodes = 512, activation = ‘relu’)
- Optimizer = adam
- Loss = ‘binary cross entropy’
- Metric = accuracy

In our final model, we will fine-tune the weights of the layers in the last two blocks of our pre-trained VGG-19 model. We will also introduce the concept of image augmentation. The idea behind image augmentation is exactly as the name sounds. We load in existing images from our training dataset and apply some image transformation operations to them, such as rotation, shearing, translation, zooming, and so on, to produce new, altered versions of existing images. Due to these random transformations, we don't get the same images each time. We will leverage an excellent utility called ImageDataGenerator in keras that can help build image augmentors.

Benchmark

For benchmark model we will try to achieve the following results :-

1.	Accuracy	>=80%
2.	F1 Score	>=85%
3.	Recall	>=90%
4.	Precision	>=80%

CNNs are excellent image classifier and can obtain accuracy upto 95%. Thus setting above benchmark are far below the expectation of CNNs models. But since we will be working with basic CNN models we are setting the values low and easily achievable. These results will be based on actual image dataset provide on Kaggle

- https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria

METHODOLOGY

Data Preprocessing

To build deep learning models, we need training data, but we also need to test the model's performance on unseen data. We will use a 60:10:30 split for train, validation, and test datasets, respectively. We will leverage the train and validation datasets during training and check the performance of the model on the test dataset.To build deep learning models, we need training data, but we also need to test the model's performance on unseen data. We will use a 60:10:30 split for train, validation, and test datasets, respectively. We will leverage the train and validation datasets during training and check the performance of the model on the test dataset.

#Code

```
from sklearn.model_selection import train_test_split
from collections import Counter

train_files, test_files, train_labels, test_labels = train_test_split(files_df['filename'].values,
                                                    files_df['label'].values,
                                                    test_size=0.3, random_state=42)
train_files, val_files, train_labels, val_labels = train_test_split(train_files,
                                                  train_labels,
                                                  test_size=0.1, random_state=42)

print(train_files.shape, val_files.shape, test_files.shape)
print('Train:', Counter(train_labels), '\nVal:', Counter(val_labels), '\nTest:', Counter(test_labels))
```

 # Output

```
(17361,) (1929,) (8268,)
Train: Counter({'healthy': 8734, 'malaria': 8627})
Val: Counter({'healthy': 970, 'malaria': 959})
Test: Counter({'malaria': 4193, 'healthy': 4075})
```

The images will not be of equal dimensions because blood smears and cell images vary based on the human, the test method, and the orientation of the photo. Let's get some summary statistics of our training dataset to determine the optimal image dimensions.

#Code

```
import cv2
from concurrent import futures
import threading

def get_img_shape_parallel(idx, img, total_imgs):
    if idx % 5000 == 0 or idx == (total_imgs - 1):
        print('{}: working on img num: {}'.format(threading.current_thread().name,
                                    idx))
    return cv2.imread(img).shape

ex = futures.ThreadPoolExecutor(max_workers=None)
data_inp = [(idx, img, len(train_files)) for idx, img in enumerate(train_files)]
print('Starting Img shape computation:')
train_img_dims_map = ex.map(get_img_shape_parallel,
                        [record[0] for record in data_inp],
                        [record[1] for record in data_inp],
                        [record[2] for record in data_inp])
train_img_dims = list(train_img_dims_map)
print('Min Dimensions:', np.min(train_img_dims, axis=0))
print('Avg Dimensions:', np.mean(train_img_dims, axis=0))
print('Median Dimensions:', np.median(train_img_dims, axis=0))
print('Max Dimensions:', np.max(train_img_dims, axis=0))
```

 # Output

```
Starting Img shape computation:
ThreadPoolExecutor-0_0: working on img num: 0
ThreadPoolExecutor-0_17: working on img num: 5000
ThreadPoolExecutor-0_15: working on img num: 10000
ThreadPoolExecutor-0_1: working on img num: 15000
ThreadPoolExecutor-0_7: working on img num: 17360
Min Dimensions: [46 46  3]
Avg Dimensions: [132.77311215 132.45757733   3.]
Median Dimensions: [130. 130.   3.]
Max Dimensions: [385 394   3]
```

We apply parallel processing to speed up the image-read operations and, based on the summary statistics, we will resize each image to 125x125 pixels. Let's load up all of our images and resize them to these fixed dimensions.

#Code

```
IMG_DIMS = (125, 125)

def get_img_data_parallel(idx, img, total_imgs):
    if idx % 5000 == 0 or idx == (total_imgs - 1):
        print('{}: working on img num: {}'.format(threading.current_thread().name,
                                                  idx))
    img = cv2.imread(img)
    img = cv2.resize(img, dsize=IMG_DIMS,
                     interpolation=cv2.INTER_CUBIC)
    img = np.array(img, dtype=np.float32)
    return img

ex = futures.ThreadPoolExecutor(max_workers=None)
train_data_inp = [(idx, img, len(train_files)) for idx, img in enumerate(train_files)]
val_data_inp = [(idx, img, len(val_files)) for idx, img in enumerate(val_files)]
test_data_inp = [(idx, img, len(test_files)) for idx, img in enumerate(test_files)]

print('Loading Train Images:')
train_data_map = ex.map(get_img_data_parallel,
                        [record[0] for record in train_data_inp],
                        [record[1] for record in train_data_inp],
                        [record[2] for record in train_data_inp])
train_data = np.array(list(train_data_map))

print('\nLoading Validation Images:')
val_data_map = ex.map(get_img_data_parallel,
                      [record[0] for record in val_data_inp],
                      [record[1] for record in val_data_inp],
                      [record[2] for record in val_data_inp])
val_data = np.array(list(val_data_map))

print('\nLoading Test Images:')
test_data_map = ex.map(get_img_data_parallel,
                       [record[0] for record in test_data_inp],
                       [record[1] for record in test_data_inp],
                       [record[2] for record in test_data_inp])
```

```
test_data = np.array(list(test_data_map))

train_data.shape, val_data.shape, test_data.shape
```

 # Output

```
Loading Train Images:
ThreadPoolExecutor-1_0: working on img num: 0
ThreadPoolExecutor-1_12: working on img num: 5000
ThreadPoolExecutor-1_6: working on img num: 10000
ThreadPoolExecutor-1_10: working on img num: 15000
ThreadPoolExecutor-1_3: working on img num: 17360

Loading Validation Images:
ThreadPoolExecutor-1_13: working on img num: 0
ThreadPoolExecutor-1_18: working on img num: 1928

Loading Test Images:
ThreadPoolExecutor-1_5: working on img num: 0
ThreadPoolExecutor-1_19: working on img num: 5000
ThreadPoolExecutor-1_8: working on img num: 8267
((17361, 125, 125, 3), (1929, 125, 125, 3), (8268, 125, 125, 3))
```

Before can we start training our models, we must set up some basic configuration settings.

#Code

```
BATCH_SIZE = 64
NUM_CLASSES = 2
EPOCHS = 25
INPUT_SHAPE = (125, 125, 3)

train_imgs_scaled = train_data / 255.
val_imgs_scaled = val_data / 255.
# encode text category labels
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
le.fit(train_labels)
train_labels_enc = le.transform(train_labels)
val_labels_enc = le.transform(val_labels)
print(train_labels[:6], train_labels_enc[:6])
```

 # Output

['malaria' 'malaria' 'malaria' 'healthy' 'healthy' 'malaria'] [1 1 1 0 0 1]

Implementation

The implementation process can be split into the following stages:-

1. Creating model architecture
2. Training the model
3. Plotting the performance of the model with :-
    a. Accuracy and Validation accuracy
    b. Train loss and Validation loss
4. Saving the model

Model 1: CNN from scratch

Our first malaria detection model will build and train a basic CNN from scratch. First, define our model architecture.

#Code : Creating model architecture

```
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D

from keras.layers import Dropout, Flatten, Dense

from keras.models import Sequential


model_basic = Sequential()

model_basic.add(Conv2D(filters = 32,padding = 'same',activation = 'relu',kernel_size = 3,
input_shape = INPUT_SHAPE))

model_basic.add(MaxPooling2D(pool_size = 2))

model_basic.add(Conv2D(filters = 64,padding = 'same',activation = 'relu',kernel_size = 3))

model_basic.add(MaxPooling2D(pool_size = 2))

model_basic.add(Conv2D(filters = 128,padding = 'same',activation = 'relu',kernel_size = 3))

model_basic.add(MaxPooling2D(pool_size = 2))
```

```
model_basic.add(Flatten())
model_basic.add(Dense(512, activation = 'relu'))
model_basic.add(Dropout(rate = 0.3))
model_basic.add(Dense(512, activation = 'relu'))
model_basic.add(Dropout(rate = 0.3))
model_basic.add(Dense(1, activation = 'sigmoid'))

#compiling the model_basic
model_basic.compile(optimizer='adam',
                    loss='binary_crossentropy',
                    metrics=['accuracy'])
#Model Summary
model_basic.summary()
```

#Output

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 125, 125, 32)      896
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 62, 62, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 62, 62, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 31, 31, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 31, 31, 128)       73856
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 15, 15, 128)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 28800)             0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               14746112
```

_____

dropout_1 (Dropout) (None, 512) 0

_____

dense_2 (Dense) (None, 512) 262656

_____

dropout_2 (Dropout) (None, 512) 0

_____

dense_3 (Dense) (None, 1) 513

==========

Total params: 15,102,529

Trainable params: 15,102,529

Non-trainable params: 0

_____

Based on the architecture in this code, our CNN model has three convolution and pooling layers, followed by two dense layers, and dropouts for regularization. Let's train our model.

#Code : Training the model

```
from keras.callbacks import ModelCheckpoint


checkpointer = ModelCheckpoint(filepath = 'CNN.Malaria.ImageProcessing.Weights.basic.T1',
verbose = 1, save_best_only = True)


history_basic = model_basic.fit(x=train_imgs_scaled, y=train_labels_enc,verbose = 1,
                                shuffle = True,batch_size=BATCH_SIZE,epochs=EPOCHS,
                                validation_data=(val_imgs_scaled, val_labels_enc),
                                callbacks = [checkpointer])
```

#Output

Train on 17361 samples, validate on 1929 samples

Epoch 1/25

17344/17361 [============================>.] - ETA: 0s - loss: 0.4691 - acc: 0.7489Epoch 00001: val_loss improved from inf to 0.18041, saving model to CNN.Malaria.ImageProcessing.Weights.basic.T1

17361/17361 [==============================] - 41s 2ms/step - loss: 0.4689 - acc: 0.7490 - val_loss: 0.1804 - val_acc: 0.9419

Epoch 2/25

17344/17361 [============================>.] - ETA: 0s - loss: 0.1539 - acc: 0.9497Epoch 00002: val_loss improved from 0.18041 to 0.14135, saving model to CNN.Malaria.ImageProcessing.Weights.basic.T1

17361/17361 [==============================] - 37s 2ms/step - loss: 0.1544 - acc: 0.9497 - val_loss: 0.1413 - val_acc: 0.9575

Epoch 25/25

17344/17361 [============================>.] - ETA: 0s - loss: 0.0056 - acc: 0.9983Epoch 00025: val_loss did not improve

17361/17361 [==============================] - 37s 2ms/step - loss: 0.0056 - acc: 0.9983 - val_loss: 0.3609 - val_acc: 0.9502

Lets plot the performance!

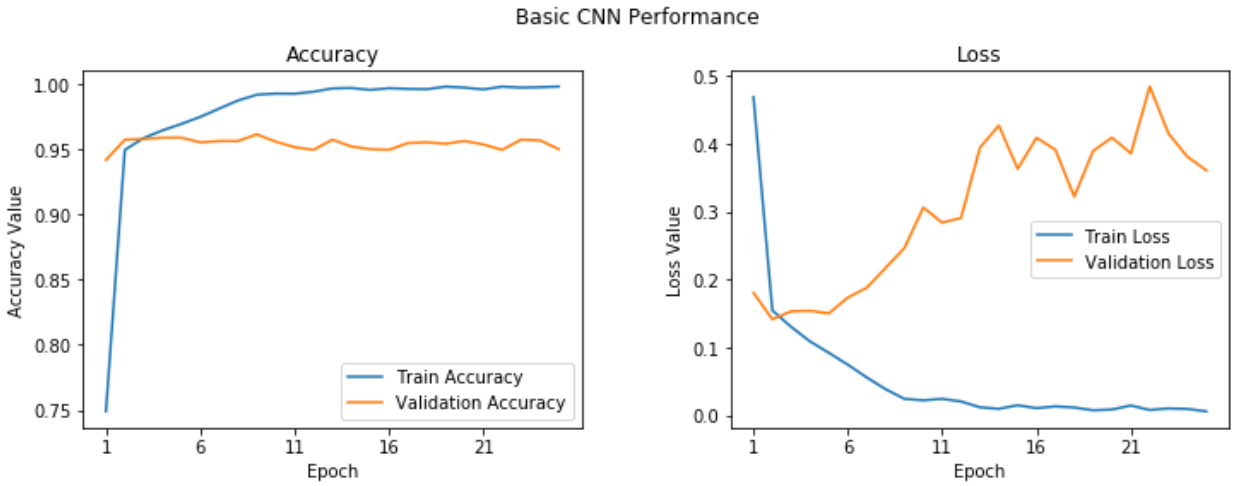
#Code : Plotting the performance

```
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
t = f.suptitle('Basic CNN Performance', fontsize=12)
f.subplots_adjust(top=0.85, wspace=0.3)

max_epoch = len(history_basic.history['acc'])+1
epoch_list = list(range(1,max_epoch))
ax1.plot(epoch_list, history_basic.history['acc'], label='Train Accuracy')
ax1.plot(epoch_list, history_basic.history['val_acc'], label='Validation Accuracy')
ax1.set_xticks(np.arange(1, max_epoch, 5))
ax1.set_ylabel('Accuracy Value')
ax1.set_xlabel('Epoch')
ax1.set_title('Accuracy')
l1 = ax1.legend(loc="best")

ax2.plot(epoch_list, history_basic.history['loss'], label='Train Loss')
ax2.plot(epoch_list, history_basic.history['val_loss'], label='Validation Loss')
ax2.set_xticks(np.arange(1, max_epoch, 5))
ax2.set_ylabel('Loss Value')
ax2.set_xlabel('Epoch')
```

ax2.set_title('Loss')

l2 = ax2.legend(loc="best")

#Output



We can see after the second epoch that things don't seem to improve a whole lot overall. Calculate accuracy and validation accuracy.

#Code

accuracy_basic = np.max(history_basic.history['acc'])*100

validation_accuracy_basic = np.max(history_basic.history['val_acc'])*100

print("Training Accuracy of basic CNN model : %.2f"%accuracy_basic+"%")

print("Validation accuracy of basic CNN model : %.2f"%validation_accuracy_basic+"%")

#Output

Training Accuracy of basic CNN model : 99.83%
Validation accuracy of basic CNN model : 96.16%

#Code: Saving the model

model_basic.save('basic_cnn.h5')

Model 2: Pre-trained model as a feature extractor

For building this model, we will leverage Keras to load up the VGG-19 model and freeze the convolution blocks so we can use them as an image feature extractor. We will plug in our own dense layers at the end to perform the classification task.

#Code : Creating the model

```
from keras.applications.vgg19 import VGG19

from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D

from keras.layers import Dropout, Flatten, Dense

from keras import Model


vgg = VGG19(include_top=False, weights='imagenet', input_shape=INPUT_SHAPE)


vgg.trainable = False

# Freeze the layers

for layer in vgg.layers:

    layer.trainable = False


base_vgg = vgg

base_out = base_vgg.output

pool_out = Flatten()(base_out)

hidden1 = Dense(512, activation='relu')(pool_out)

drop1 = Dropout(rate=0.3)(hidden1)

hidden2 = Dense(512, activation='relu')(drop1)

drop2 = Dropout(rate=0.3)(hidden2)


out = Dense(1, activation='sigmoid')(drop2)


model_pre_trained = Model(inputs=base_vgg.input, outputs=out)
```

```
model_pre_trained.compile(optimizer = 'RMSprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model_pre_trained.summary()
```

#Output

```
Downloading data from
https://github.com/fchollet/deep-learning-models/releases/download/v0.1/vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5

80142336/80134624 [==============================] - 1s 0us/step

_________________________________________________________________

Layer (type)                 Output Shape              Param #

=================================================================

input_1 (InputLayer)         (None, 125, 125, 3)       0

_________________________________________________________________

block1_conv1 (Conv2D)        (None, 125, 125, 64)      1792

_________________________________________________________________

block1_conv2 (Conv2D)        (None, 125, 125, 64)      36928

_________________________________________________________________

block1_pool (MaxPooling2D)   (None, 62, 62, 64)        0

_________________________________________________________________

block2_conv1 (Conv2D)        (None, 62, 62, 128)       73856

_________________________________________________________________

block2_conv2 (Conv2D)        (None, 62, 62, 128)       147584

_________________________________________________________________

block2_pool (MaxPooling2D)   (None, 31, 31, 128)       0

_________________________________________________________________

block3_conv1 (Conv2D)        (None, 31, 31, 256)       295168

_________________________________________________________________

block3_conv2 (Conv2D)        (None, 31, 31, 256)       590080

_________________________________________________________________
```

block3_conv3 (Conv2D) (None, 31, 31, 256) 590080

---

block3_conv4 (Conv2D) (None, 31, 31, 256) 590080

---

block3_pool (MaxPooling2D) (None, 15, 15, 256) 0

---

block4_conv1 (Conv2D) (None, 15, 15, 512) 1180160

---

block4_conv2 (Conv2D) (None, 15, 15, 512) 2359808

---

block4_conv3 (Conv2D) (None, 15, 15, 512) 2359808

---

block4_conv4 (Conv2D) (None, 15, 15, 512) 2359808

---

block4_pool (MaxPooling2D) (None, 7, 7, 512) 0

---

block5_conv1 (Conv2D) (None, 7, 7, 512) 2359808

---

block5_conv2 (Conv2D) (None, 7, 7, 512) 2359808

---

block5_conv3 (Conv2D) (None, 7, 7, 512) 2359808

---

block5_conv4 (Conv2D) (None, 7, 7, 512) 2359808

---

block5_pool (MaxPooling2D) (None, 3, 3, 512) 0

---

flatten_2 (Flatten) (None, 4608) 0

---

dense_4 (Dense) (None, 512) 2359808

---

dropout_3 (Dropout) (None, 512) 0

_____

dense_5 (Dense) (None, 512) 262656

_____

dropout_4 (Dropout) (None, 512) 0

_____

dense_6 (Dense) (None, 1) 513

==========

Total params: 22,647,361

Trainable params: 2,622,977

Non-trainable params: 20,024,384

_____

It is evident from this output that we have a lot of layers in our model and we will be using the frozen layers of the VGG-19 model as feature extractors only. You can use the following code to verify how many layers in our model are indeed trainable and how many total layers are present in our network.

#Code

```
print("Total Layers:", len(model.layers))
print("Total trainable layers:",
    sum([1 for l in model.layers if l.trainable]))
```

 # Output

Total Layers: 28

Total trainable layers: 6

#Code : Training the model

```
from keras.callbacks import ModelCheckpoint
checkpointer = ModelCheckpoint(filepath = 'CNN.Malaria.ImageProcessing.Weights.VGG19.PreTrained.T2',
                               verbose = 1, save_best_only = True)
history_pre_trained = model_pre_trained.fit(x = train_imgs_scaled,
```

```
                    y = train_labels_enc, verbose = 1,  shuffle = True,
                    batch_size=BATCH_SIZE,epochs=EPOCHS,
                    validation_data=(val_imgs_scaled, val_labels_enc),
                    callbacks = [checkpointer])
```

#Output

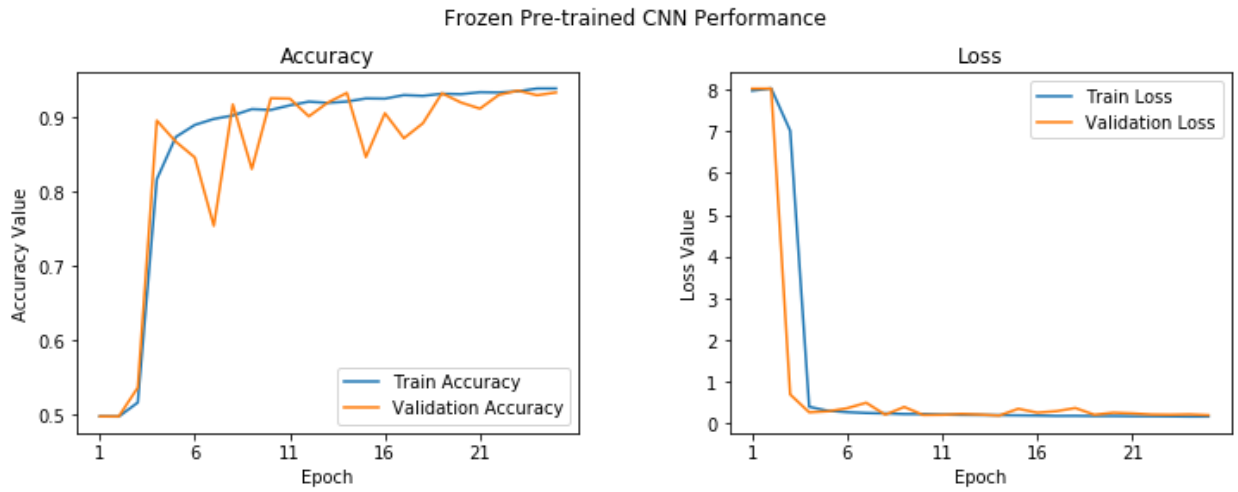
```
Train on 17361 samples, validate on 1929 samples

Epoch 1/25

17344/17361 [============================>.] - ETA: 0s - loss: 7.9679 - acc: 0.4971Epoch 00001: val_loss improved from inf to 8.01665, saving model to CNN.Malaria.ImageProcessing.Weights.VGG19.PreTrained.T2

17361/17361 [==============================] - 151s 9ms/step - loss: 7.9665 - acc: 0.4971 - val_loss: 8.0166 - val_acc: 0.4971

Epoch 14/25

17344/17361 [============================>.] - ETA: 0s - loss: 0.2030 - acc: 0.9219Epoch 00014: val_loss improved from 0.20791 to 0.19463, saving model to CNN.Malaria.ImageProcessing.Weights.VGG19.PreTrained.T2

17361/17361 [==============================] - 147s 8ms/step - loss: 0.2028 - acc: 0.9220 - val_loss: 0.1946 - val_acc: 0.9336

Epoch 25/25

17344/17361 [============================>.] - ETA: 0s - loss: 0.1701 - acc: 0.9396Epoch 00025: val_loss did not improve

17361/17361 [==============================] - 147s 8ms/step - loss: 0.1700 - acc: 0.9396 - val_loss: 0.2038 - val_acc: 0.9342
```

#Code : Plotting the performance

```
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
t = f.suptitle('Frozen Pre-trained CNN Performance', fontsize=12)
f.subplots_adjust(top=0.85, wspace=0.3)


max_epoch = len(history_pre_trained.history['acc'])+1
epoch_list = list(range(1,max_epoch))
ax1.plot(epoch_list, history_pre_trained.history['acc'], label='Train Accuracy')
ax1.plot(epoch_list, history_pre_trained.history['val_acc'], label='Validation Accuracy')
ax1.set_xticks(np.arange(1, max_epoch, 5))
ax1.set_ylabel('Accuracy Value')
ax1.set_xlabel('Epoch')
ax1.set_title('Accuracy')
```

```
l1 = ax1.legend(loc="best")


ax2.plot(epoch_list, history_pre_trained.history['loss'], label='Train Loss')
ax2.plot(epoch_list, history_pre_trained.history['val_loss'], label='Validation Loss')
ax2.set_xticks(np.arange(1, max_epoch, 5))
ax2.set_ylabel('Loss Value')
ax2.set_xlabel('Epoch')
ax2.set_title('Loss')
l2 = ax2.legend(loc="best")
```

#Output



This shows that our model is not overfitting as much as our basic CNN model, but the performance is slightly less than our basic CNN model. Let's save this model for future evaluation. Calculating the accuracy and validation accuracy.

#Code

```
accuracy_pre_trained = np.max(history_pre_trained.history['acc'])*100
validation_accuracy_pre_trained = np.max(history_pre_trained.history['val_acc'])*100
print("Accuracy of pre_trained CNN model : %.2f"%accuracy_pre_trained+"%")
print("Validation accuracy of pre_trained CNN model :
%.2f"%validation_accuracy_pre_trained+"%")
```

#Output

Accuracy of pre_trained CNN model : 93.96%

Validation accuracy of pre_trained CNN model : 93.68%

#Code : Saving the model

```
model.save('vgg_frozen.h5')
```

Model 3: Fine-tuned pre-trained model with image augmentation

Image Augmentation

#Code

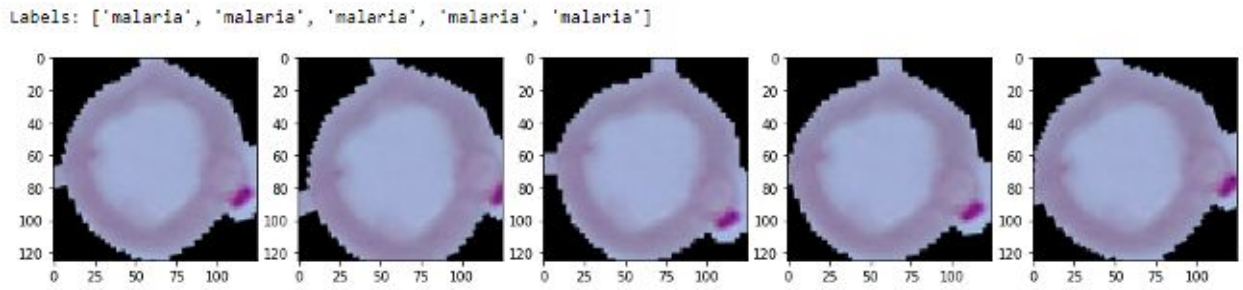
```
train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255,
                                                                zoom_range=0.05,
                                                                rotation_range=25,
                                                                width_shift_range=0.05,
                                                                height_shift_range=0.05,
                                                                shear_range=0.05, horizontal_flip=True,
                                                                fill_mode='nearest')


val_datagen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255)

# build image augmentation generators

train_generator = train_datagen.flow(train_data, train_labels_enc, batch_size=BATCH_SIZE,
shuffle=True)

val_generator = val_datagen.flow(val_data, val_labels_enc, batch_size=BATCH_SIZE,
shuffle=False)
```

Let's look at some sample results from a batch of image augmentation transforms.

#Code

```
img_id = 0

sample_generator = train_datagen.flow(train_data[img_id:img_id+1],
train_labels[img_id:img_id+1],
                                      batch_size=1)

sample = [next(sample_generator) for i in range(0,5)]

fig, ax = plt.subplots(1,5, figsize=(16, 6))

print('Labels:', [item[1][0] for item in sample])

l = [ax[i].imshow(sample[i][0][0]) for i in range(0,5)]
```

#Output



You can clearly see the slight variations of our images in the preceding output. We will now build our deep learning models, making sure the last two blocks of the VGG-19 model are trainable.

#Code : Creating the model

```
from keras import optimizers


vgg = VGG19(include_top=False, weights='imagenet',
                                input_shape=INPUT_SHAPE)
# Freeze the layers
vgg.trainable = True
set_trainable = False
for layer in vgg.layers:
    if layer.name in ['block5_conv1', 'block4_conv1']:
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
base_vgg = vgg
base_out = base_vgg.output
pool_out = Flatten()(base_out)
hidden1 = Dense(512, activation='relu')(pool_out)
drop1 = Dropout(rate=0.3)(hidden1)
hidden2 = Dense(512, activation='relu')(drop1)
```

```
drop2 = Dropout(rate=0.3)(hidden2)
out = Dense(1, activation='sigmoid')(drop2)
model_fine_tuned = Model(inputs=base_vgg.input, outputs=out)
RMSprop = optimizers.RMSprop(lr=1e-5)
model_fine_tuned.compile(optimizer = RMSprop,
              loss='binary_crossentropy',
              metrics=['accuracy'])
model_fine_tuned.summary()
print("Total Layers:", len(model_fine_tuned.layers))
print("Total trainable layers:", sum([1 for l in model_fine_tuned.layers if l.trainable]))
from keras import optimizers

vgg = VGG19(include_top=False, weights='imagenet',
                                input_shape=INPUT_SHAPE)

# Freeze the layers
vgg.trainable = True

set_trainable = False
for layer in vgg.layers:
    if layer.name in ['block5_conv1', 'block4_conv1']:
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False



base_vgg = vgg
```

```
base_out = base_vgg.output
pool_out = Flatten()(base_out)
hidden1 = Dense(512, activation='relu')(pool_out)
drop1 = Dropout(rate=0.3)(hidden1)
hidden2 = Dense(512, activation='relu')(drop1)
drop2 = Dropout(rate=0.3)(hidden2)

out = Dense(1, activation='sigmoid')(drop2)

model_fine_tuned = Model(inputs=base_vgg.input, outputs=out)

RMSprop = optimizers.RMSprop(lr=1e-5)

model_fine_tuned.compile(optimizer = RMSprop,
            loss='binary_crossentropy',
            metrics=['accuracy'])
model_fine_tuned.summary()
print("Total Layers:", len(model_fine_tuned.layers))
print("Total trainable layers:", sum([1 for l in model_fine_tuned.layers if l.trainable]))
```

#Output

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 125, 125, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 125, 125, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 125, 125, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 62, 62, 64)        0
_________________________________________________________________
```

block2_conv1 (Conv2D) (None, 62, 62, 128) 73856

---

block2_conv2 (Conv2D) (None, 62, 62, 128) 147584

---

block2_pool (MaxPooling2D) (None, 31, 31, 128) 0

---

block3_conv1 (Conv2D) (None, 31, 31, 256) 295168

---

block3_conv2 (Conv2D) (None, 31, 31, 256) 590080

---

block3_conv3 (Conv2D) (None, 31, 31, 256) 590080

---

block3_conv4 (Conv2D) (None, 31, 31, 256) 590080

---

block3_pool (MaxPooling2D) (None, 15, 15, 256) 0

---

block4_conv1 (Conv2D) (None, 15, 15, 512) 1180160

---

block4_conv2 (Conv2D) (None, 15, 15, 512) 2359808

---

block4_conv3 (Conv2D) (None, 15, 15, 512) 2359808

---

block4_conv4 (Conv2D) (None, 15, 15, 512) 2359808

---

block4_pool (MaxPooling2D) (None, 7, 7, 512) 0

---

block5_conv1 (Conv2D) (None, 7, 7, 512) 2359808

---

block5_conv2 (Conv2D) (None, 7, 7, 512) 2359808

---

block5_conv3 (Conv2D) (None, 7, 7, 512) 2359808

---

block5_conv4 (Conv2D) (None, 7, 7, 512) 2359808

---

block5_pool (MaxPooling2D) (None, 3, 3, 512) 0

_____

flatten_3 (Flatten) (None, 4608) 0

_____

dense_7 (Dense) (None, 512) 2359808

_____

dropout_5 (Dropout) (None, 512) 0

_____

dense_8 (Dense) (None, 512) 262656

_____

dropout_6 (Dropout) (None, 512) 0

_____

dense_9 (Dense) (None, 1) 513

===

Total params: 22,647,361

Trainable params: 20,321,793

Non-trainable params: 2,325,568

_____

Total Layers: 28

Total trainable layers: 16

We reduce the learning rate in our model since we don't want to make to large weight updates to the pre-trained layers when fine-tuning. The model's training process will be slightly different since we are using data generators, so we will be leveraging the fit_generator function.

#Code : Training the model

```
from keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(filepath = 'CNN.Malaria.ImageProcessing.Weights.VGG19.FineTuned.T3',
                               verbose = 1, save_best_only = True)

train_steps_per_epoch = train_generator.n // train_generator.batch_size

val_steps_per_epoch = val_generator.n // val_generator.batch_size

history_fine_tuned = model_fine_tuned.fit_generator(train_generator,
                                                    steps_per_epoch=train_steps_per_epoch,
                                                    epochs=EPOCHS,
                                                    validation_data=val_generator,
                                                    validation_steps=val_steps_per_epoch,
                                                verbose=1,callbacks = [checkpointer])
```

#Output

Epoch 1/25

270/271 [============================>.] - ETA: 0s - loss: 0.2298 - acc: 0.9091Epoch 00001: val_loss improved from inf to 0.13384, saving model to CNN.Malaria.ImageProcessing.Weights.VGG19.FineTuned.T3

271/271 [==============================] - 254s 936ms/step - loss: 0.2295 - acc: 0.9092 - val_loss: 0.1338 - val_acc: 0.9552

Epoch 3/25

270/271 [============================>.] - ETA: 0s - loss: 0.1221 - acc: 0.9589Epoch 00003: val_loss improved from 0.13384 to 0.12608, saving model to CNN.Malaria.ImageProcessing.Weights.VGG19.FineTuned.T3

271/271 [==============================] - 251s 926ms/step - loss: 0.1224 - acc: 0.9588 - val_loss: 0.1261 - val_acc: 0.9594

Epoch 6/25

270/271 [============================>.] - ETA: 0s - loss: 0.1062 - acc: 0.9638Epoch 00006: val_loss improved from 0.12608 to 0.10953, saving model to CNN.Malaria.ImageProcessing.Weights.VGG19.FineTuned.T3

271/271 [==============================] - 251s 926ms/step - loss: 0.1061 - acc: 0.9638 - val_loss: 0.1095 - val_acc: 0.9635

Epoch 14/25

270/271 [============================>.] - ETA: 0s - loss: 0.0857 - acc: 0.9714Epoch 00014: val_loss improved from 0.10953 to 0.10902, saving model to CNN.Malaria.ImageProcessing.Weights.VGG19.FineTuned.T3

271/271 [==============================] - 251s 925ms/step - loss: 0.0860 - acc: 0.9712 - val_loss: 0.1090 - val_acc: 0.9661

Epoch 25/25

270/271 [============================>.] - ETA: 0s - loss: 0.0825 - acc: 0.9726Epoch 00025: val_loss did not improve

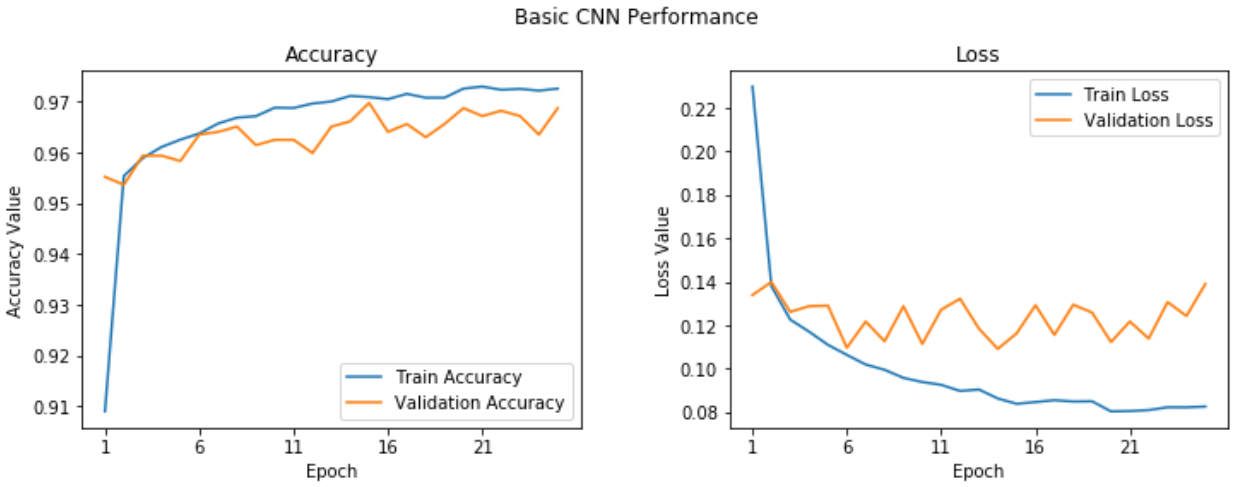
271/271 [==============================] - 250s 923ms/step - loss: 0.0823 - acc: 0.9727 - val_loss: 0.1389 - val_acc: 0.9688

#Code : Plotting the performance of model

```
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

t = f.suptitle('Basic CNN Performance', fontsize=12)

f.subplots_adjust(top=0.85, wspace=0.3)

max_epoch = len(history_fine_tuned.history['acc'])+1

epoch_list = list(range(1,max_epoch))

ax1.plot(epoch_list, history_fine_tuned.history['acc'], label='Train Accuracy')

ax1.plot(epoch_list, history_fine_tuned.history['val_acc'], label='Validation Accuracy')

ax1.set_xticks(np.arange(1, max_epoch, 5))

ax1.set_ylabel('Accuracy Value')

ax1.set_xlabel('Epoch')

ax1.set_title('Accuracy')

l1 = ax1.legend(loc="best")

ax2.plot(epoch_list, history_fine_tuned.history['loss'], label='Train Loss')
```

```
ax2.plot(epoch_list, history_fine_tuned.history['val_loss'], label='Validation Loss')

ax2.set_xticks(np.arange(1, max_epoch, 5))

ax2.set_ylabel('Loss Value')

ax2.set_xlabel('Epoch')

ax2.set_title('Loss')

l2 = ax2.legend(loc="best")
```

#Output



#Code : Calculation the accuracy and validation accuracy

```
accuracy_fine_tuned = np.max(history_fine_tuned.history['acc'])*100

validation_accuracy_fine_tuned = np.max(history_fine_tuned.history['val_acc'])*100


print("Accuracy of fine_tuned CNN model : %.2f"%accuracy_fine_tuned+"%")

print("Validation accuracy of fine_tuned CNN model : %.2f"%validation_accuracy_fine_tuned+"%")
```

#Output

Accuracy of fine_tuned CNN model : 97.30%
Validation accuracy of fine_tuned CNN model : 96.98%

#Code : Saving the model

```
model_fine_tuned.save('vgg_finetuned.h5')
```

This looks to be the best model yet. It gives us a validation accuracy of almost 96.5% and, based on the training accuracy, it doesn't look like our model is overfitting as much as our first model. This can be verified with the following learning curves.

Refinement

We will evaluate the three models we built in the training phase by making predictions with them on the data from our test dataset. model_evaluation_utils module is created to evaluate the performance of the deep learning models with classification metrics. Steps followed for evaluation are :-

- Scale test data
- Load saved deep learning models
- Make prediction on test data
- Leverage model_evaluation_utils to check the performance of models

#Code : Scaling the data

```
test_imgs_scaled = test_data / 255.

test_imgs_scaled.shape, test_labels.shape
```

#Output
((8268, 125, 125, 3), (8268,))

#Code : Loading the models

```
from keras import models

basic_cnn = models.load_model('./basic_cnn.h5')

vgg_frz = models.load_model('./vgg_frozen.h5')

vgg_ft = models.load_model('./vgg_finetuned.h5')
```

#Code : Making prediction on test data

```
basic_cnn_preds = basic_cnn.predict(test_imgs_scaled, batch_size=512)

vgg_frz_preds = vgg_frz.predict(test_imgs_scaled, batch_size=512)

vgg_ft_preds = vgg_ft.predict(test_imgs_scaled, batch_size=512)

basic_cnn_pred_labels = le.inverse_transform([1 if pred > 0.5 else 0
                                              for pred in basic_cnn_preds.ravel()])

vgg_frz_pred_labels = le.inverse_transform([1 if pred > 0.5 else 0
                                            for pred in vgg_frz_preds.ravel()])

vgg_ft_pred_labels = le.inverse_transform([1 if pred > 0.5 else 0
                                           for pred in vgg_ft_preds.ravel()])
```

#Code : Evaluation

```
import model_evaluation_utils as meu

import pandas as pd

basic_cnn_metrics = meu.get_metrics(true_labels=test_labels,
predicted_labels=basic_cnn_pred_labels)

vgg_frz_metrics = meu.get_metrics(true_labels=test_labels,
predicted_labels=vgg_frz_pred_labels)

vgg_ft_metrics = meu.get_metrics(true_labels=test_labels, predicted_labels=vgg_ft_pred_labels)

pd.DataFrame([basic_cnn_metrics, vgg_frz_metrics, vgg_ft_metrics],

        index=['Basic CNN', 'VGG-19 Frozen', 'VGG-19 Fine-tuned'])
```

	Accuracy	F1 Score:	Precision:	Recall
Basic CNN	0.9507	0.9507	0.9507	0.9507
VGG-19 Frozen	0.9347	0.9347	0.9349	0.9347
VGG-19 Fine-tuned	0.9653	0.9653	0.9654	0.9653

Observations

Initial basic CNN model , we have achieved score as follows :

- Training Accuracy = 0.9983
- Validation Accuracy = 0.9616
- Testing Accuracy = 0.9507
- F1 Score = 0.9507
- Precision = 0.9507
- Recall = 0.9507

These score achieve are very good figures considering our benchmark. But as it can be noticed from the training accuracy of the model that it is overfitting. Model is very robust with an F1 score of 0.9507. Recall and precision values are also very high then our expectations.

Next VGG-19 Frozen CNN model is not overfitting as basic model as observed in scores below :

- Training Accuracy = 0.9396
- Validation Accuracy = 0.9368
- Testing Accuracy = 0.9347
- F1 Score = 0.9347
- Precision = 0.9349
- Recall = 0.9347

This shows that our model is not overfitting as much as our basic CNN model, but the performance is slightly less than our basic CNN model.

Final model scores are best among all the models. Model is giving a test accuracy score of 0.9653, which is better than both of the previous models. Model is very slightly over fitting but rest of the score are better.

- Training Accuracy = 0.9730
- Validation Accuracy = 0.9698
- Testing Accuracy = 0.9653
- F1 Score = 0.9653
- Precision = 0.9654
- Recall = 0.9653

Thus using transfer learning model VGG-19 with trainable layer and image augmentation is the best and final solution for malaria detection problem.

RESULTS

Model Evaluation and Validation

Model classification report and predicted confusion matrix are created for each of the models.

1. Basic model

#Code

```
meu.display_model_performance_metrics(true_labels=test_labels,
                                      predicted_labels=basic_cnn_pred_labels,
                                      classes=list(set(test_labels)))
```

#Output

Model Performance metrics:

Model Classification report:

---

	precision	recall	f1-score	support
healthy	0.95	0.95	0.95	4075
malaria	0.95	0.95	0.95	4193
avg / total	0.95	0.95	0.95	8268

Prediction Confusion Matrix:

	Predicted:	
Actual:	healthy	malaria
healthy	3867	208
malaria	200	3993

2. VGG-19 Frozen

#Code

```
meu.display_model_performance_metrics(true_labels=test_labels,
                                      predicted_labels=vgg_frz_pred_labels,
                                      classes=list(set(test_labels)))
```

#Output

Model Performance metrics:

Model Classification report:

---

	precision	recall	f1-score	support
healthy	0.92	0.94	0.93	4075
malaria	0.94	0.93	0.93	4193
avg / total	0.93	0.93	0.93	8268

Prediction Confusion Matrix:

Actual:	Predicted: healthy	malaria
healthy	3849	226
malaria	314	3879

3. VGG-19 Fine tuned

#Code

```
meu.display_model_performance_metrics(true_labels=test_labels,
                                      predicted_labels=vgg_ft_pred_labels,
                                      classes=list(set(test_labels)))
```

#Output

Model Performance metrics:

Model Classification report:

---

	precision	recall	f1-score	support
healthy	0.96	0.97	0.97	4075
malaria	0.97	0.96	0.97	4193
avg / total	0.97	0.97	0.97	8268

Prediction Confusion Matrix:

	Predicted:	
Actual:	healthy	malaria
healthy	3968	107
malaria	180	4013

Final model has provided great results as depicted above. With implementation of image and augmentation and training entire VGG-19 model has worked excellently. Model has been trained with unseen testing data and considering validation accuracy and testing accuracy we can state that model has been generalized and can be implemented in practical use. Also high F1-score of 97% vouch for robustness of the model. As stated that we implemented image augmentation of training data, we can say that tweaking the data as improved the performance of model.

Justification

Final result are far stronger than the benchmark set for evaluating model performance. We have analyzed each metric mentioned in the benchmark table.

S.No.	Metric	Benchmark	Final Model
1.	Accuracy	>=80%	96.53%
2.	F1 Score	>=85%	96.53%
3.	Recall	>=90%	96.54%
4.	Precision	>=80%	96.53%

Deep learning CNN models has provided excellent solution to Malaria detection problem. The results are extremely encouraging for the implementation of AI in the field of medical science. Such models can be easily implemented in practical use with Android and iOS apps to provide quick and cheap detection of malaria and reduce manual effort and cost of the process in countries where people can not afford proper health care facilities and lack qualified professionals. As there is always room for improvement depicted in above 3 models where we achieved improving result with each attempt, we can state that better and deeper model can be created to obtain even better results.

CONCLUSION

Free-Form Visualization

Confusion matrix visualized above in model visualization and evaluation depict the metrics and prove the excellent performance of models. We have also displayed the accuracies curve and loss function of the models earlier to display learning rate of models. We have discussed all the mentioned visualization thoroughly throughout the report.

Reflection

The entire process can be explained as an implementation of CNN models as image classifier for image datasets. Datasets involves two types of images, malaria infected and uninfected. Each model is provided with a training set which is a combination of both types of images. CNN learns the patterns and differences in between both types of images and classify the testing image dataset into infected and uninfected categories. The interesting aspect of the project is amazing ability of transfer learning of CNNs model. As we know that these algorithms are designed on the basis of human brain. As the human brain has the ability of passing the knowledge, these models also hold with amazing ability. This truly amazing and a great achievement of computer science. The difficult aspect of the project was training the models. Each model took a large amount of time in training. Even though we used good GPUs, each model took around an hour or two to complete 25 epochs. So also considering the code debugging, it took hours and hours to completely train all the models.

As mentioned above the final solution or the final model is an excellent classifier and the result obtained can be implemented in practical use in mobile devices in countries where people lack proper health care facilites.

Improvement

As we developed each model and found out that with each alteration and algorithm we obtained better results. Thus it can be stated that there is still room for improvement. Other transfer learning techniques such as Resnet50 and Xception algorithms could provide us even better results. Thus overall we can say that “Yes, better results can still be achieved”.

Reference

1. https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria
2. https://ceb.nlm.nih.gov/repositories/malaria-datasets/
3. https://en.wikipedia.org/wiki/Malaria

Code for model_evaluation_utils

```
from sklearn import metrics
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.base import clone
from sklearn.preprocessing import label_binarize
from scipy import interp
from sklearn.metrics import roc_curve, auc


def get_metrics(true_labels, predicted_labels):
    return {
        'Accuracy': np.round(
                        metrics.accuracy_score(true_labels,
                                               predicted_labels),4),
        'Precision:': np.round(
                        metrics.precision_score(true_labels,
                                               predicted_labels,
                                               average='weighted'),4),
        'Recall': np.round(
                        metrics.recall_score(true_labels,
                                               predicted_labels,
                                               average='weighted'),4),
        'F1 Score:': np.round(
                        metrics.f1_score(true_labels,
                                               predicted_labels,
                                               average='weighted'),4)
    }
```

```
def display_metrics(true_labels, predicted_labels):
    print('Accuracy:', np.round(
                        metrics.accuracy_score(true_labels,
                                               predicted_labels),4))
    print('Precision:', np.round(
                        metrics.precision_score(true_labels,
                                               predicted_labels,
                                               average='weighted'),4))
    print('Recall:', np.round(
                        metrics.recall_score(true_labels,
                                               predicted_labels,
                                               average='weighted'),4))
    print('F1 Score:', np.round(
                        metrics.f1_score(true_labels,
                                               predicted_labels,
                                               average='weighted'),4))


def train_predict_model(classifier,
                        train_features, train_labels,
                        test_features, test_labels):
    # build model
    classifier.fit(train_features, train_labels)
    # predict using model
    predictions = classifier.predict(test_features)
    return predictions


def display_confusion_matrix(true_labels, predicted_labels, classes=[1,0]):


    total_classes = len(classes)
    level_labels = [total_classes*[0], list(range(total_classes))]
```

```
    cm = metrics.confusion_matrix(y_true=true_labels, y_pred=predicted_labels,
                                  labels=classes)
    cm_frame = pd.DataFrame(data=cm,
                            columns=pd.MultiIndex(levels=[['Predicted:'], classes],
                                                  labels=level_labels),
                            index=pd.MultiIndex(levels=[['Actual:'], classes],
                                                labels=level_labels))
    print(cm_frame)

def display_confusion_matrix_pretty(true_labels, predicted_labels, classes=[1,0]):

    total_classes = len(classes)
    level_labels = [total_classes*[0], list(range(total_classes))]

    cm = metrics.confusion_matrix(y_true=true_labels, y_pred=predicted_labels,
                                  labels=classes)
    cm_frame = pd.DataFrame(data=cm,
                            columns=pd.MultiIndex(levels=[['Predicted:'], classes],
                                                  labels=level_labels),
                            index=pd.MultiIndex(levels=[['Actual:'], classes],
                                                labels=level_labels))
    return cm_frame

def display_classification_report(true_labels, predicted_labels, classes=[1,0]):
    report = metrics.classification_report(y_true=true_labels,
                                           y_pred=predicted_labels,
                                           labels=classes)
    print(report)
```

```
def display_model_performance_metrics(true_labels, predicted_labels, classes=[1,0]):
    print('Model Performance metrics:')
    print('-'*30)
    get_metrics(true_labels=true_labels, predicted_labels=predicted_labels)
    print('\nModel Classification report:')
    print('-'*30)
    display_classification_report(true_labels=true_labels, predicted_labels=predicted_labels,
                                  classes=classes)
    print('\nPrediction Confusion Matrix:')
    print('-'*30)
    display_confusion_matrix(true_labels=true_labels, predicted_labels=predicted_labels,
                             classes=classes)


def plot_model_decision_surface(clf, train_features, train_labels,
                                plot_step=0.02, cmap=plt.cm.RdYlBu,
                                markers=None, alphas=None, colors=None):


    if train_features.shape[1] != 2:
        raise ValueError("X_train should have exactly 2 columnns!")


    x_min, x_max = train_features[:, 0].min() - plot_step, train_features[:, 0].max() + plot_step
    y_min, y_max = train_features[:, 1].min() - plot_step, train_features[:, 1].max() + plot_step
    xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                         np.arange(y_min, y_max, plot_step))


    clf_est = clone(clf)
    clf_est.fit(train_features,train_labels)
    if hasattr(clf_est, 'predict_proba'):
        Z = clf_est.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:,1]
    else:
```

```
        Z = clf_est.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    cs = plt.contourf(xx, yy, Z, cmap=cmap)


    le = LabelEncoder()
    y_enc = le.fit_transform(train_labels)
    n_classes = len(le.classes_)
    plot_colors = ''.join(colors) if colors else [None] * n_classes
    label_names = le.classes_
    markers = markers if markers else [None] * n_classes
    alphas = alphas if alphas else [None] * n_classes
    for i, color in zip(range(n_classes), plot_colors):
        idx = np.where(y_enc == i)
        plt.scatter(train_features[idx, 0], train_features[idx, 1], c=color,
                    label=label_names[i], cmap=cmap, edgecolors='black',
                    marker=markers[i], alpha=alphas[i])
    plt.legend()
    plt.show()


def plot_model_roc_curve(clf, features, true_labels, label_encoder=None, class_names=None):


    ## Compute ROC curve and ROC area for each class
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    if hasattr(clf, 'classes_'):
        class_labels = clf.classes_
    elif label_encoder:
        class_labels = label_encoder.classes_
    elif class_names:
```

```
    class_labels = class_names
else:
    raise ValueError('Unable to derive prediction classes, please specify class_names!')
n_classes = len(class_labels)
y_test = label_binarize(true_labels, classes=class_labels)
if n_classes == 2:
    if hasattr(clf, 'predict_proba'):
        prob = clf.predict_proba(features)
        y_score = prob[:, prob.shape[1]-1]
    elif hasattr(clf, 'decision_function'):
        prob = clf.decision_function(features)
        y_score = prob[:, prob.shape[1]-1]
    else:
        raise AttributeError("Estimator doesn't have a probability or confidence scoring system!")

    fpr, tpr, _ = roc_curve(y_test, y_score)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label='ROC curve (area = {0:0.2f})'
                         ''.format(roc_auc),
             linewidth=2.5)

elif n_classes > 2:
    if hasattr(clf, 'predict_proba'):
        y_score = clf.predict_proba(features)
    elif hasattr(clf, 'decision_function'):
        y_score = clf.decision_function(features)
    else:
        raise AttributeError("Estimator doesn't have a probability or confidence scoring system!")

    for i in range(n_classes):
```

```
    fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

## Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test.ravel(), y_score.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

## Compute macro-average ROC curve and ROC area
# First aggregate all false positive rates
all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))
# Then interpolate all ROC curves at this points
mean_tpr = np.zeros_like(all_fpr)
for i in range(n_classes):
    mean_tpr += interp(all_fpr, fpr[i], tpr[i])
# Finally average it and compute AUC
mean_tpr /= n_classes
fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

## Plot ROC curves
plt.figure(figsize=(6, 4))
plt.plot(fpr["micro"], tpr["micro"],
         label='micro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["micro"]), linewidth=3)

plt.plot(fpr["macro"], tpr["macro"],
         label='macro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["macro"]), linewidth=3)
```

```
    for i, label in enumerate(class_labels):
        plt.plot(fpr[i], tpr[i], label='ROC curve of class {0} (area = {1:0.2f})'
                                 ''.format(label, roc_auc[i]),
                 linewidth=2, linestyle=':')
else:
    raise ValueError('Number of classes should be atleast 2 or more')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```