

Amirkabir University of Technology

(Tehran Polytechnic)

Department of Industrial Engineering & Management Systems

**Mask Detection Using Convolutional Neural Networks**

By:

Pedram Peiro Asfia - 9825006

Professor:

Dr. Marzieh Zarrinbal Masouleh

Course:

Artificial Intelligence

Jan 2023

**Contents**

[1. Problem Statement 5](#_Toc125276468)

[2. Introduction 6](#_Toc125276469)

[3. Relevant Libraries 6](#_Toc125276470)

[4. Extracting the Annotations of The Images 7](#_Toc125276471)

[5. Data Visualization & Analysis of The Target Label 8](#_Toc125276472)

[6. Images Identification with Target Class 9](#_Toc125276473)

[7. Image Preprocessing 12](#_Toc125276474)

[8. Splitting Dataset into Train/Dev/Test Sets 16](#_Toc125276475)

[9. Implementing CNN Models 18](#_Toc125276476)

[9.1. Basic Models Implementations 19](#_Toc125276477)

[9.1.1. First Model 19](#_Toc125276478)

[9.1.2. Second Model 25](#_Toc125276479)

[9.1.3. Third Model 31](#_Toc125276480)

[9.2. Transfer Learning 36](#_Toc125276481)

[9.2.1. MobileNetV2 36](#_Toc125276482)

[9.2.2. VGG-19 40](#_Toc125276483)

[9.2.3. ResNet-50 45](#_Toc125276484)

[9.2.4. Inception-V3 48](#_Toc125276485)

[10. Misclassified Images 52](#_Toc125276486)

**Figures**

[Figure 1 - Sample record 1 5](#_Toc125276405)

[Figure 2 - Sample record 2 5](#_Toc125276406)

[Figure 3 - Classes and their frequency 8](#_Toc125276407)

[Figure 4 - Data visualization of the frequency of classes 9](#_Toc125276408)

[Figure 5 - Raw images without faces identified 9](#_Toc125276409)

[Figure 6 - Faces identified and labeled 10](#_Toc125276410)

[Figure 7 - Random image with faces identified No. 1 11](#_Toc125276411)

[Figure 8 - Random image with faces identified No. 2 12](#_Toc125276412)

[Figure 9 - Random image with faces identified No. 3 12](#_Toc125276413)

[Figure 10 - Cropped images No. 1 14](#_Toc125276414)

[Figure 11 - Cropped images No. 2 15](#_Toc125276415)

[Figure 12 - Cropped images No. 3 15](#_Toc125276416)

[Figure 13 – Data frame of images path and label 16](#_Toc125276417)

[Figure 14 - Instances of the test set 18](#_Toc125276418)

[Figure 15 - First model's architecture 19](#_Toc125276419)

[Figure 16 - First model's dot format architecture 20](#_Toc125276420)

[Figure 17 - First model's output 22](#_Toc125276421)

[Figure 18 - Training stage for the first model 22](#_Toc125276422)

[Figure 19 - Train and dev set loss for each epoch 23](#_Toc125276423)

[Figure 20 - Measurements for the first model on the train set 23](#_Toc125276424)

[Figure 21 - Measurements for the first model on the dev set 24](#_Toc125276425)

[Figure 22 - Measurements for the first model on the test set 24](#_Toc125276426)

[Figure 23 – Second model's architecture 25](#_Toc125276427)

[Figure 24 - Second model's dot format architecture 26](#_Toc125276428)

[Figure 25 - Second model's output 27](#_Toc125276429)

[Figure 26 - Training stage for the second model 28](#_Toc125276430)

[Figure 27 - Train and dev set loss for each epoch 29](#_Toc125276431)

[Figure 28 - Measurements for the second model on the train set 29](#_Toc125276432)

[Figure 29 - Measurements for the second model on the dev set 30](#_Toc125276433)

[Figure 30 - Measurements for the second model on the test set 30](#_Toc125276434)

[Figure 31 – Third model's architecture 31](#_Toc125276435)

[Figure 32 - Third model's dot format architecture 32](#_Toc125276436)

[Figure 33 - Second model's output 33](#_Toc125276437)

[Figure 34 - Training stage for the second model 33](#_Toc125276438)

[Figure 35 - Train and dev set loss for each epoch 34](#_Toc125276439)

[Figure 36 - Measurements for the third model on the train set 35](#_Toc125276440)

[Figure 37 - Measurements for the third model on the dev set 35](#_Toc125276441)

[Figure 38 - Measurements for the third model on the test set 36](#_Toc125276442)

[Figure 39 - MobileNetV2 architecture 37](#_Toc125276443)

[Figure 40 – MobileNetV2 output 38](#_Toc125276444)

[Figure 41 - Training stage for the MobileNetV2 model 38](#_Toc125276445)

[Figure 42 - Train and test set loss for each epoch 39](#_Toc125276446)

[Figure 43 - Measurements for MobileNetV2 on train set 40](#_Toc125276447)

[Figure 44 - Measurements for MobileNetV2 on the test set 40](#_Toc125276448)

[Figure 45 – VGG-19 architecture 41](#_Toc125276449)

[Figure 46 - VGG-19 output 42](#_Toc125276450)

[Figure 47 - Training stage for the VGG-19 model 43](#_Toc125276451)

[Figure 48 - Train and test set loss for each epoch 44](#_Toc125276452)

[Figure 49 - Measurements for VGG-19 on train set 44](#_Toc125276453)

[Figure 50 - Measurements for VGG-19 on the test set 45](#_Toc125276454)

[Figure 51 – ResNet-50 architecture 46](#_Toc125276455)

[Figure 52 – ResNet-50 output 46](#_Toc125276456)

[Figure 53 - Training stage for the ResNet-50 model 47](#_Toc125276457)

[Figure 54 - Train and test set loss for each epoch 47](#_Toc125276458)

[Figure 55 - Measurements for ResNet-50 on train set 48](#_Toc125276459)

[Figure 56 - Measurements for ResNet-50 on the test set 48](#_Toc125276460)

[Figure 57 – Inception-V3 architecture 49](#_Toc125276461)

[Figure 58 – Inception-V3 output 50](#_Toc125276462)

[Figure 59 - Training stage for the Inception-V3 model 50](#_Toc125276463)

[Figure 60 - Train and test set loss for each epoch 51](#_Toc125276464)

[Figure 61 - Measurements for Inception-V3 on train set 52](#_Toc125276465)

[Figure 62 - Measurements for Inception-V3 on the test set 52](#_Toc125276466)

[Figure 63 - Misclassified images based on the best model 53](#_Toc125276467)

# 1. Problem Statement

In this project, the objective is to detect whether a person is wearing a face mask or not. The application of building such a system is in public places which a lot of people walk through. This way we (as the government) can understand whether folk are listening to medical pieces of advice or ignoring them. Therefore, based on the current circumstances, what steps should be taken to tackle the pandemic and as a consequence of people’s neglect, how fast the illness will spread.

Consequently, we might be able to take some advantageous steps to be readier to confront the issue and grapple with it.

Now we’ll take a look at some samples of the dataset.



Figure 1 - Sample record 1



Figure 2 - Sample record 2

# 2. Introduction

A Convolutional Neural Network, also known as CNN or ConvNet, is a class of neural networks that specializes in processing data that has a grid-like topology, such as an image. A digital image is a binary representation of visual data. It contains a series of pixels arranged in a grid-like fashion that contains pixel values to denote how bright and what color each pixel should be. The human brain processes a huge amount of information the second we see an image.

Each neuron works in its own receptive field and is connected to other neurons in a way that they cover the entire visual field. Just as each neuron responds to stimuli only in the restricted region of the visual field called the receptive field in the biological vision system, each neuron in a CNN processes data only in its receptive field as well. The layers are arranged in such a way that they detect simpler patterns first (lines, curves, etc.) and more complex patterns (faces, objects, etc.) further along. By using a CNN, one can enable sight to computers.

# 3. Relevant Libraries

Numerous libraries were utilized in this project which can be seen below:

* NumPy
* Pandas
* Matplotlib
* Seaborn
* Collections
* OS
* CV2
* SciPy
* Xmltodict
* Torch
* Torchvision
* PIL
* Tensorflow
* Keras
* Glob
* Scikit-learn

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib.patches as mpatches

import seaborn as sns

from collections import Counter

import os

from os import listdir

import cv2

from scipy.spatial import distance

import xmltodict

import torch

from torchvision import transforms,models

from torch.utils.data import Dataset,DataLoader

from PIL import Image

import tensorflow as tf

import sys

import torch.optim as optim

import glob

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import classification\_report, confusion\_matrix

import tensorflow as tf

from tensorflow.keras import Sequential, models

from tensorflow.keras.layers import Flatten, Dense, Conv2D, MaxPool2D

from keras.preprocessing.image import ImageDataGenerator

import tensorflow\_addons as tfa

import tensorflow.keras.layers as tfl

from keras.callbacks import ModelCheckpoint, EarlyStopping

# 4. Extracting the Annotations of The Images

In order to identify the faces in each of the images and their state (wearing a mask or not), we have to use XML files and map them to their corresponding images.

First, we append the path of images and annotations to some lists:

img\_names=[]

xml\_names=[]

for dirname, \_ , filenames in os.walk('Data'):

    for filename in filenames:

        if os.path.join(dirname, filename)[-3:]!="xml":

            img\_names.append(filename)

        else:

            xml\_names.append(filename)

Then we’ll check the classes and their frequency:

path\_annotations="Data/annotations/"

listing=[]

for i in img\_names[:]:

    with open(path\_annotations+i[:-4]+".xml") as fd:

        doc=xmltodict.parse(fd.read())

    temp=doc["annotation"]["object"]

    if type(temp)==list:

        for i in range(len(temp)):

            listing.append(temp[i]["name"])

    else:

        listing.append(temp["name"])

Items = Counter(listing).keys()

values = Counter(listing).values()

print(Items,'\n',values)

Which culminates in the following result:



Figure 3 - Classes and their frequency

# 5. Data Visualization & Analysis of The Target Label

Now it’s time for a little bit of DataViz (also known as data visualization) to see the proportion of each class:

fig, (ax1, ax2) = plt.subplots(1, 2, figsize =(14,6))

plt.style.use('seaborn')

ax1.pie(list(values) , labels=list(Items), shadow=True , startangle=90 ,autopct='%1.1f%%' , wedgeprops={'edgecolor': 'black'})

ax2.bar(list(Items) , height = list(values) , width = .5)

fig.suptitle('Analysis of Data Label' , fontsize=16)

plt.tight\_layout()

plt.show()

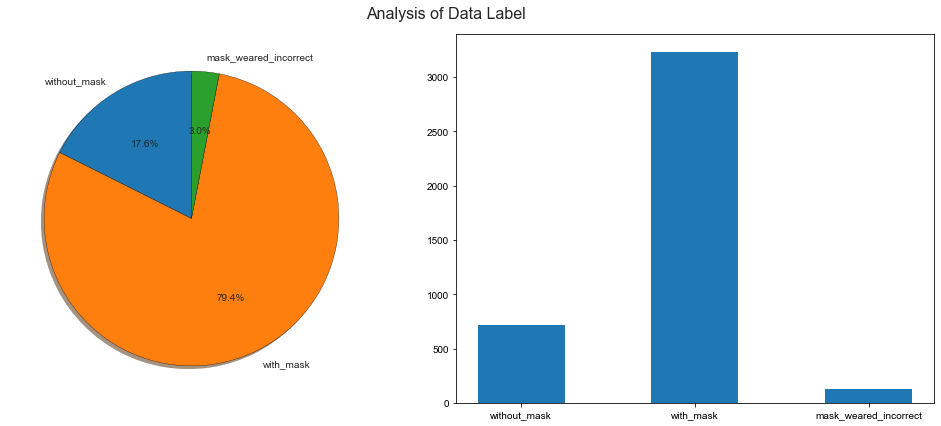


Figure 4 - Data visualization of the frequency of classes

# 6. Images Identification with Target Class

As it was alluded before, 3 classes exist and they are specified with different colors cascading faces:

* Red: without\_mask
* Green: with\_mask
* Yellow: mask\_weared\_incorrect

If we take a look at the images provided in “Data/images”, we’ll see that people’s faces aren’t specified:



Figure 5 - Raw images without faces identified

And we have to use XML files to identify them:

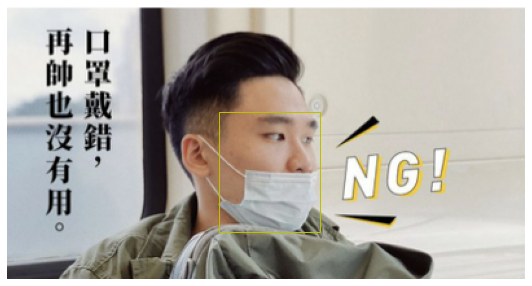


Figure 6 - Faces identified and labeled

To do this we use the XML files saved in “Data/annotation”:

path\_image="Data/images/"

def face\_cas(img):

    with open(path\_annotations+img[:-4]+".xml") as fd:

        doc=xmltodict.parse(fd.read())

    image=plt.imread(os.path.join(path\_image+img))

    fig,ax=plt.subplots(1,figsize = (10,5))

    ax.axis("off")

    temp=doc["annotation"]["object"]

    if isinstance(temp,list):

        for i in range(len(temp)):

            # with\_mask

            if temp[i]["name"]=="with\_mask":

                x,y,w,h=list(map(int,temp[i]["bndbox"].values()))

                mpatch=mpatches.Rectangle((x,y),w-x,h-y,linewidth=1, edgecolor='g',facecolor="none",lw=2,)

                ax.add\_patch(mpatch)

                rx, ry = mpatch.get\_xy()

                ax.annotate("with\_mask", (rx, ry), color='green', weight='bold', fontsize=10, ha='left', va='baseline')

            # without\_mask

            if temp[i]["name"]=="without\_mask":

                x,y,w,h=list(map(int,temp[i]["bndbox"].values()))

                mpatch=mpatches.Rectangle((x,y),w-x,h-y,linewidth=1, edgecolor='r',facecolor="none",lw=2,)

                ax.add\_patch(mpatch)

                rx, ry = mpatch.get\_xy()

                ax.annotate("without\_mask", (rx, ry), color='red', weight='bold', fontsize=10, ha='left', va='baseline')

            # mask\_weared\_incorrect

            if temp[i]["name"]=="mask\_weared\_incorrect":

                x,y,w,h=list(map(int,temp[i]["bndbox"].values()))

                mpatch=mpatches.Rectangle((x,y),w-x,h-y,linewidth=1, edgecolor='y',facecolor="none",lw=2,)

                ax.add\_patch(mpatch)

                rx, ry = mpatch.get\_xy()

                ax.annotate("mask\_weared\_incorrect", (rx, ry), color='yellow', weight='bold', fontsize=10, ha='left', va='baseline')

    else:

        x,y,w,h=list(map(int,temp["bndbox"].values()))

        edgecolor={"with\_mask":"g","without\_mask":"r","mask\_weared\_incorrect":"y"}

        mpatch=mpatches.Rectangle((x,y),w-x,h-y,linewidth=1, edgecolor=edgecolor[temp["name"]],facecolor="none",)

    ax.imshow(image)

    ax.add\_patch(mpatch)

Consequently, we can check the output for 10 random images:

rand\_images = img\_names.copy()

for  i in np.random.randint(0 , np.sum(len(img\_names)) , size=10):

    face\_cas(rand\_images[i])



Figure 7 - Random image with faces identified No. 1

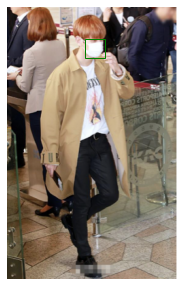


Figure 8 - Random image with faces identified No. 2



Figure 9 - Random image with faces identified No. 3

Other images can be checked in the notebook.

# 7. Image Preprocessing

Before training CNN models, we have to apply some preprocessing to the given images.

For convenience, we consider the following dictionary to describe the classes:

options = {"with\_mask":0,"without\_mask":1,"mask\_weared\_incorrect":2}

Afterward, it’s time to crop the images based on the coordinations provided in XML files to only capture the faces with their masks. The rationale behind it is that background is not an important feature in our classification and should be removed.

For this section, we use the transforms method imported from torchvision.

def dataset\_creation(image\_list):

    image\_tensor=[]

    label\_tensor=[]

    for i,j in enumerate(image\_list):

        with open(path\_annotations+j[:-4]+".xml") as fd:

            doc=xmltodict.parse(fd.read())

        if isinstance(doc["annotation"]["object"] , list):

            temp=doc["annotation"]["object"]

            for k in range(len(temp)):

                x,y,w,h=list(map(int,temp[k]["bndbox"].values()))

                label=options[temp[k]["name"]]

                image=transforms.functional.crop(Image.open(path\_image+j).convert("RGB"),y,x,h-y,w-x)

                image\_tensor.append(my\_transform(image))

                label\_tensor.append(torch.tensor(label))

        else:

            temp=doc["annotation"]["object"]

            x,y,w,h=list(map(int,temp["bndbox"].values()))

            label=options[temp["name"]]

            image=transforms.functional.crop(Image.open(path\_image+j).convert("RGB"), y,x,h-y,w-x)

            image\_tensor.append(my\_transform(image))

            label\_tensor.append(torch.tensor(label))

    final\_dataset=[[k,l] for k,l in zip(image\_tensor,label\_tensor)]

    return tuple(final\_dataset)

This function does the cropping task for us and returns a tuple of lists in which the tensor of the cropped images is in addition to their labels.

As we desire to have the images 256x256:

my\_transform=transforms.Compose([transforms.Resize((256,256)),

                                 transforms.ToTensor()])

mydataset=dataset\_creation(img\_names)

However, torch.tensor is not the datatype that we’d like to work with and tf.tensor is much more preferred. So, we change the datatype by first converting their type into a numpy array and then transforming it to tensorflow’s tensor.

mydataset = tuple([list((tf.convert\_to\_tensor(y.numpy()) for y in x)) for x in mydataset])

After that, we check the resulting images.

for i in np.random.randint(0,len(mydataset),5):

    img = tf.keras.preprocessing.image.array\_to\_img(mydataset[i][0] , data\_format="channels\_first")

    plt.imshow(img)

    plt.show()

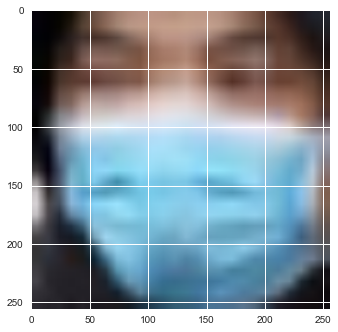


Figure 10 - Cropped images No. 1

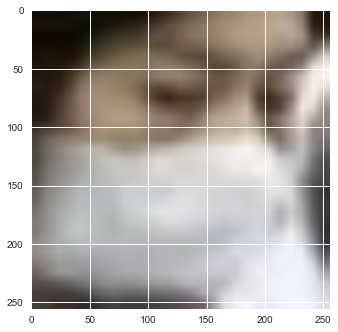


Figure 11 - Cropped images No. 2

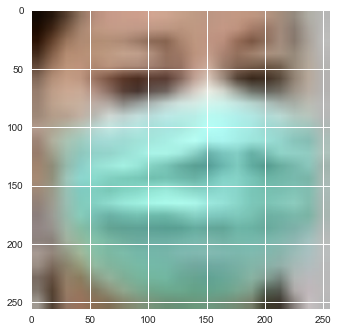


Figure 12 - Cropped images No. 3

In order to not pass all the preceding stages again and again whenever we want to try a model, we save the cropped images in the "edited pics" directory.

for i in range(len(mydataset)):

    label = int(mydataset[i][1])

    img = mydataset[i][0]

    tf.keras.utils.save\_img(f"edited pics/{i}. {label}.png" , img, data\_format="channels\_first" , scale=True)

# 8. Splitting Dataset into Train/Dev/Test Sets

To achieve this, we parse through the paths of the images and save them with their labels in a data frame:

images\_path = []

images\_label = []

# get the path/directory

folder\_dir = "edited pics"

for images in os.listdir(folder\_dir):

    # check if the image ends with png

    if (images.endswith(".png")):

        # print(images[-5:-4])

        images\_path.append(images)

        images\_label.append(images[-5:-4])

df = pd.DataFrame({'image\_path':images\_path , 'image\_label':images\_label})

df['image\_label'] = df['image\_label'].astype(int)

df.head()

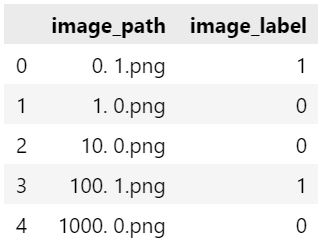


Figure 13 – Data frame of images path and label

The instances with label 2 (denoting mask\_weared\_incorrect) will be removed because we have no certain interest in them.

df = df[df['image\_label']!=2]

Next, we split the whole remaining images by the proportion of 80%, 10% & 10% for train, dev, and test sets respectively.

X\_train, X\_test, y\_train, y\_test = train\_test\_split( df['image\_path'], df['image\_label'], test\_size=0.2, random\_state=1)

X\_dev, X\_test , y\_dev, y\_test = train\_test\_split( X\_test, y\_test, test\_size=0.5, random\_state=1)

X\_train, X\_dev, and X\_test have the path of the images and not the images themselves, so we should load the images. Of course, we should pay heed that the images which were saved are in BGR form and have to be changed to RGB. Besides, the desired size of images is 256x256.

def reading\_images( X, y, path, image\_size = 256):

    X\_list = []

    y\_list = []

    for i in X.index:

        img\_array = cv2.imread(path+"/"+str(X[i]))

        img\_array = cv2.cvtColor(img\_array, cv2.COLOR\_BGR2RGB)

        new\_image\_array = cv2.resize(img\_array.astype(np.uint8), (image\_size, image\_size))

        X\_list.append(new\_image\_array)

        y\_list.append(y[i])

    return X\_list,y\_list

# Reading all the images into a list and changing the size of the image to (256,256)

path = 'edited pics'

X\_train\_arr , y\_train\_arr = reading\_images(X\_train , y\_train, path)

X\_dev\_arr , y\_dev\_arr = reading\_images(X\_dev , y\_dev, path)

X\_test\_arr , y\_test\_arr = reading\_images(X\_test , y\_test, path)

Now let’s look at the training example:

## Looking at the training samples

fig, ax = plt.subplots(2, 3, figsize=(10, 10))

for row in range(2):

    for col in range(3):

        idx = np.random.choice(np.arange(len(y\_train\_arr)))

        ax[row, col].axis("off")

        ax[row,col].imshow(X\_train\_arr[idx])

        if y\_train\_arr[idx] == 0:

            ax[row, col].set\_title("With Mask")

        else:

            ax[row, col].set\_title("Without Mask")

plt.show()

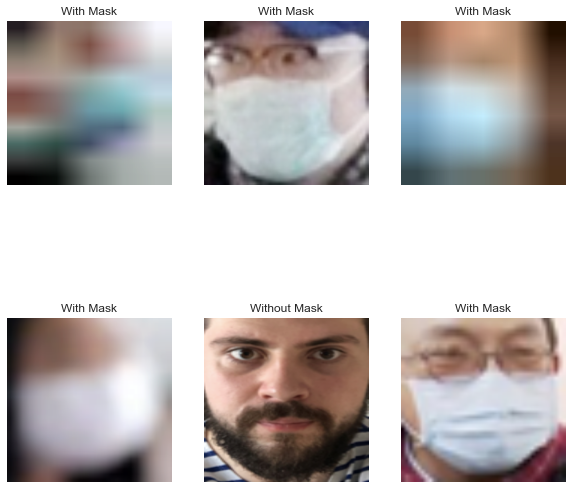


Figure 14 - Instances of the test set

Further preprocessing like converting the datatypes to numpy.array and normalizing is done:

## Converting X and y to a numpy array as Tensorflow accepts only numpy arrays

X\_train\_arr,y\_train\_arr = np.array(X\_train\_arr) , np.array(y\_train\_arr)

X\_dev\_arr,y\_dev\_arr = np.array(X\_dev\_arr) , np.array(y\_dev\_arr)

X\_test\_arr,y\_test\_arr = np.array(X\_test\_arr) , np.array(y\_test\_arr)

### Normalizing the data

X\_train\_arr, X\_dev\_arr, X\_test\_arr = X\_train\_arr/255, X\_dev\_arr/255, X\_test\_arr/255

y\_train\_arr, y\_dev\_arr, y\_test\_arr = y\_train\_arr.reshape(-1,1), y\_dev\_arr.reshape(-1,1), y\_test\_arr.reshape(-1,1)

# 9. Implementing CNN Models

In this section different CNN models are implemented, either from scratch or using base models as a transfer learning task. Because we can’t have access to tensorflow’s website properly (because it is forbidden for Iranian users), another notebook for transfer learning was created in google colab and GPU was used there.

## 9.1. Basic Models Implementations

### 9.1.1. First Model

#### 9.1.1.1. Model Implementation

Our first model contains 4 convolutional layers (with max-pooling layers following them) and 3 fully connected layers. The general architecture is as below which was created using the “visualkeras” library.

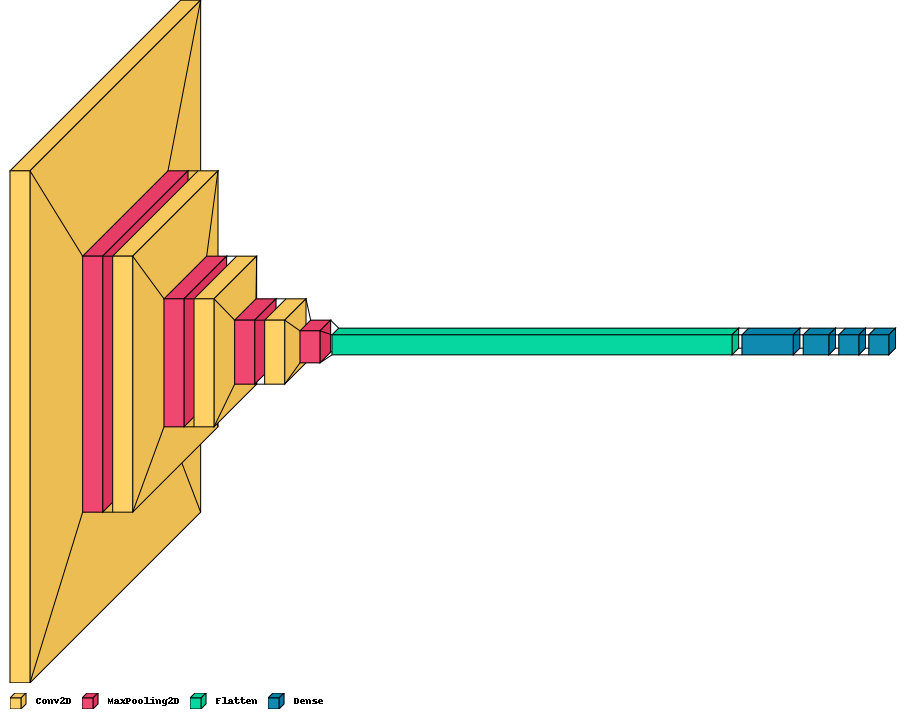


Figure 15 - First model's architecture

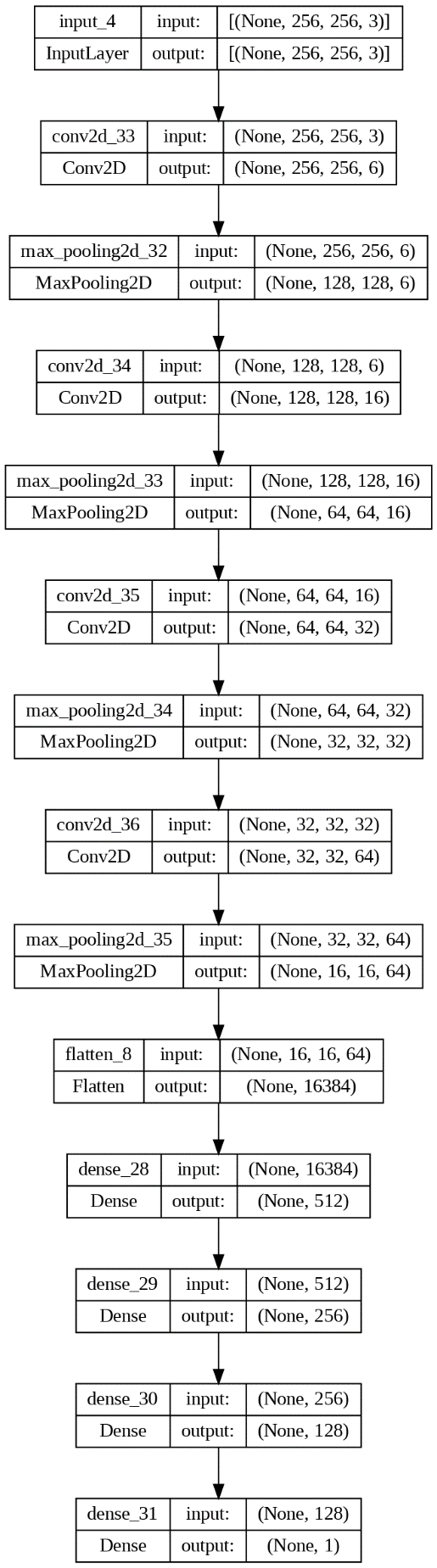


Figure 16 - First model's dot format architecture

model = Sequential([

    Conv2D(6 , kernel\_size=5 , activation = "relu" , padding='same'),

    MaxPool2D(2 , strides=2 ,padding='same'),

    Conv2D(16 , kernel\_size=5 , activation='relu' , padding='same'),

    MaxPool2D(2, strides=2, padding='same'),

    Conv2D(32 , kernel\_size=5 , activation = "relu" , padding='same'),

    MaxPool2D(2, strides=2, padding='same'),

    Conv2D(64 , kernel\_size=5 , activation = "relu" , padding='same'),

    MaxPool2D(3, strides=2, padding='same'),

    Flatten(),

    Dense(512 , activation='relu'),

    Dense(256 , activation='relu'),

    Dense(128 , activation='relu'),

    Dense(1, activation='sigmoid')

])

Then the model should be compiled with Adam as its optimizer and binary cross entropy as its loss function.

model.compile(optimizer='adam',

              loss=

              'binary\_crossentropy',

              metrics=[tfa.metrics.F1Score(num\_classes=1), 'accuracy'])

The model is then trained with a batch size equal to 32 and for 10 epochs. Although we want to run it for 10 epochs, we set an early stopping condition which may terminate the training phase sooner than expected to eschew overfitting.

The best model is saved as the name of '4CNN-3FC\_Simple.h5' and can be seen in figure 17.

save\_dir = "./results/"

model\_name = '4CNN-3FC\_Simple.h5'

model\_path = os.path.join(save\_dir, model\_name)

history\_simple\_CNN = model.fit(X\_train\_arr, y\_train\_arr,

          batch\_size=32,

          epochs=10,

          validation\_data=(X\_dev\_arr, y\_dev\_arr),

          callbacks=[

            ModelCheckpoint(model\_path, save\_best\_only=True),

            EarlyStopping(monitor='val\_loss', mode='min', patience=2, min\_delta=0.0001, verbose=2)

          ])



Figure 17 - First model's output

And the training stage is shown below:

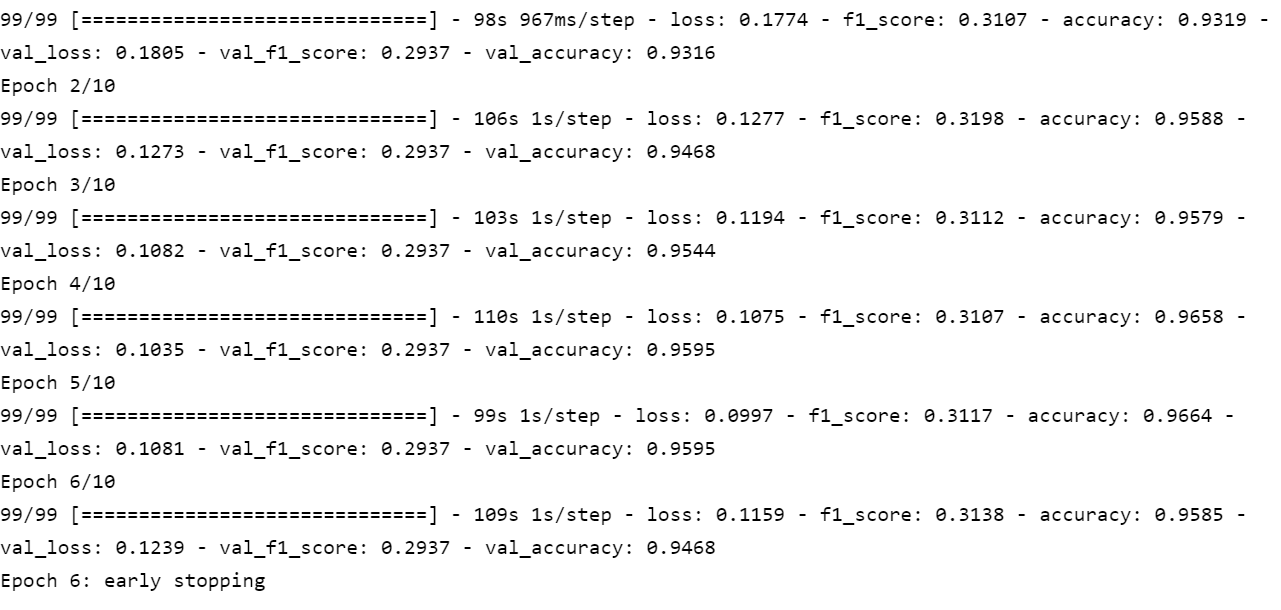


Figure 18 - Training stage for the first model

The summary section shows that the model has 8,620,441 parameters, all of which are trainable.

Additionally, the performance of the model on the train set and dev set according to each epoch is depicted below:

# plotting the metrics

fig = plt.figure(figsize=(13,6))

ax = plt.subplot(111)

ax.plot(history\_simple\_CNN.history['loss'])

ax.plot(history\_simple\_CNN.history['val\_loss'])

ax.set\_title('4CNN-3FC\_Simple loss')

ax.set\_ylabel('loss')

ax.set\_xlabel('epoch')

ax.legend(['train', 'test'], loc='upper right')

ax.grid(True)

plt.show()

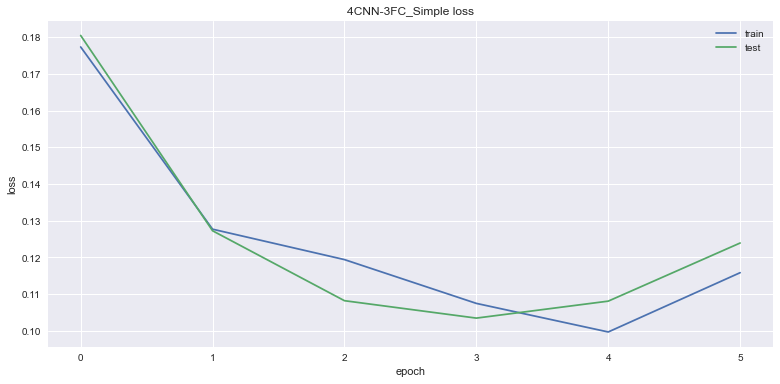


Figure 19 - Train and dev set loss for each epoch

#### 9.1.1.2. Model Assessment

We should assess the output of this model with the help of some measurements like accuracy, F1-score & etc., and also do some error analysis. Although we perform error analysis, we won’t change the structure of the model or augment data or anything else.

prediction = (model.predict(X\_train\_arr)).squeeze()

prediction = np.where(prediction>=0.5 , 1 , 0)

print(classification\_report(prediction, y\_train\_arr))

print(confusion\_matrix(prediction, y\_train\_arr))

And the same codes are applied to dev & test sets, the results are demonstrated below.

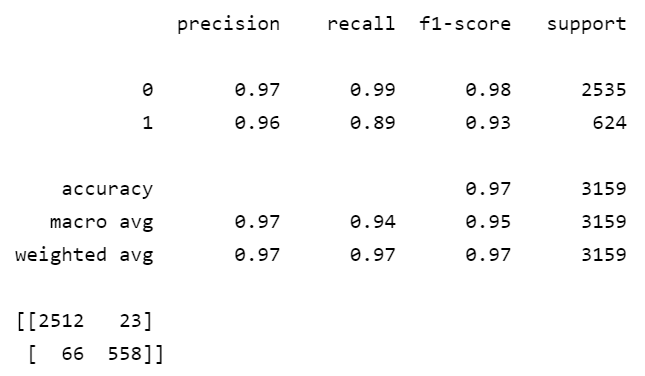


Figure 20 - Measurements for the first model on the train set

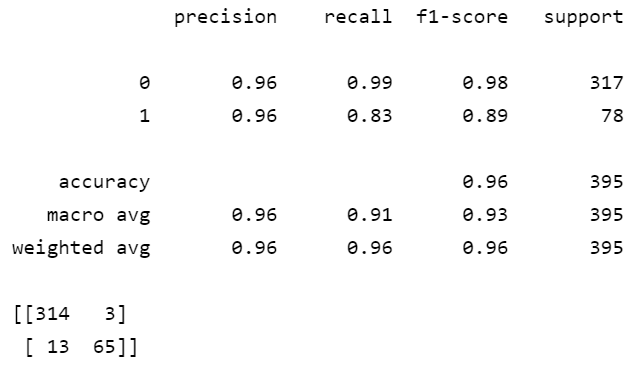


Figure 21 - Measurements for the first model on the dev set

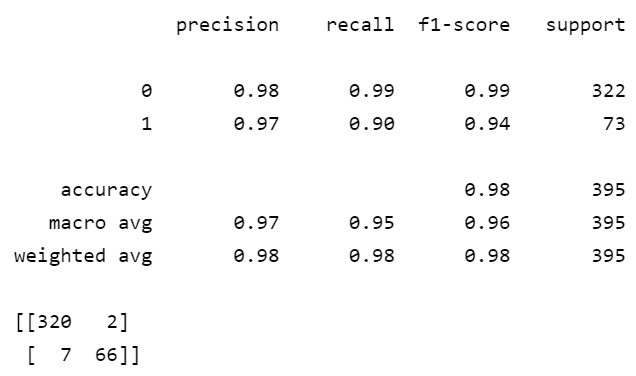


Figure 22 - Measurements for the first model on the test set

Because humans can perfectly detect whether someone has worn a mask or not, human-level performance error is equal to 0.00%.

Table 1 - Error analysis for the first model

|  |  |
| --- | --- |
| Type | Error |
| Human-level performance | 0.00% |
| Train set | 3% |
| Dev set | 4% |
| Test set | 2% |

As we terminated the model sooner than training to its full potential, it’s conspicuous that the error of the test set is low, even less than the train set itself. Of course, we have encountered the problem of bias, because the train set error is not near human-level performance error. In order to address this issue, we have the following options:

* Train bigger model
* Train longer, use better optimization algorithms
* Change the architecture/ hyperparameter tuning

### 9.1.2. Second Model

#### 9.1.2.1. Model Implementation

Our second model contains 2 convolutional layers (with max-pooling layers following them) and 3 fully connected layers. The general architecture is as below:

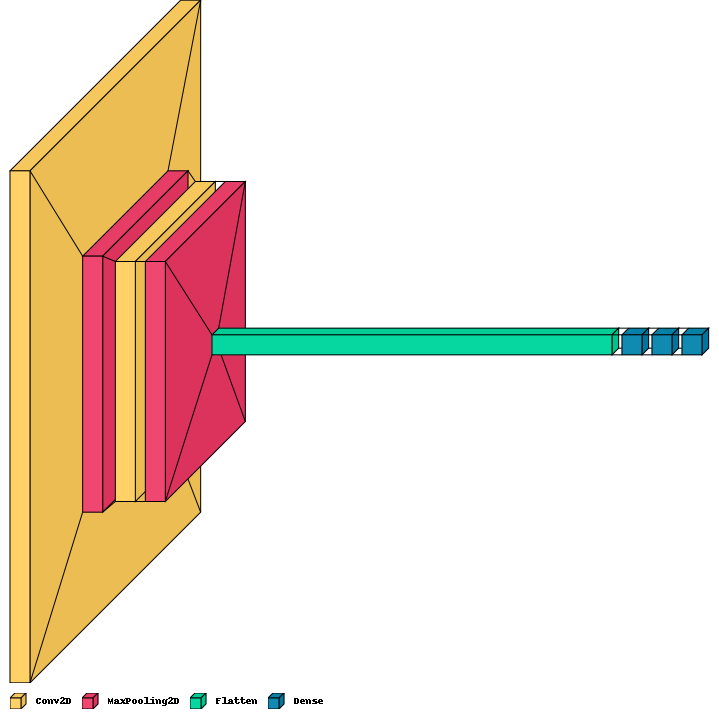


Figure 23 – Second model's architecture

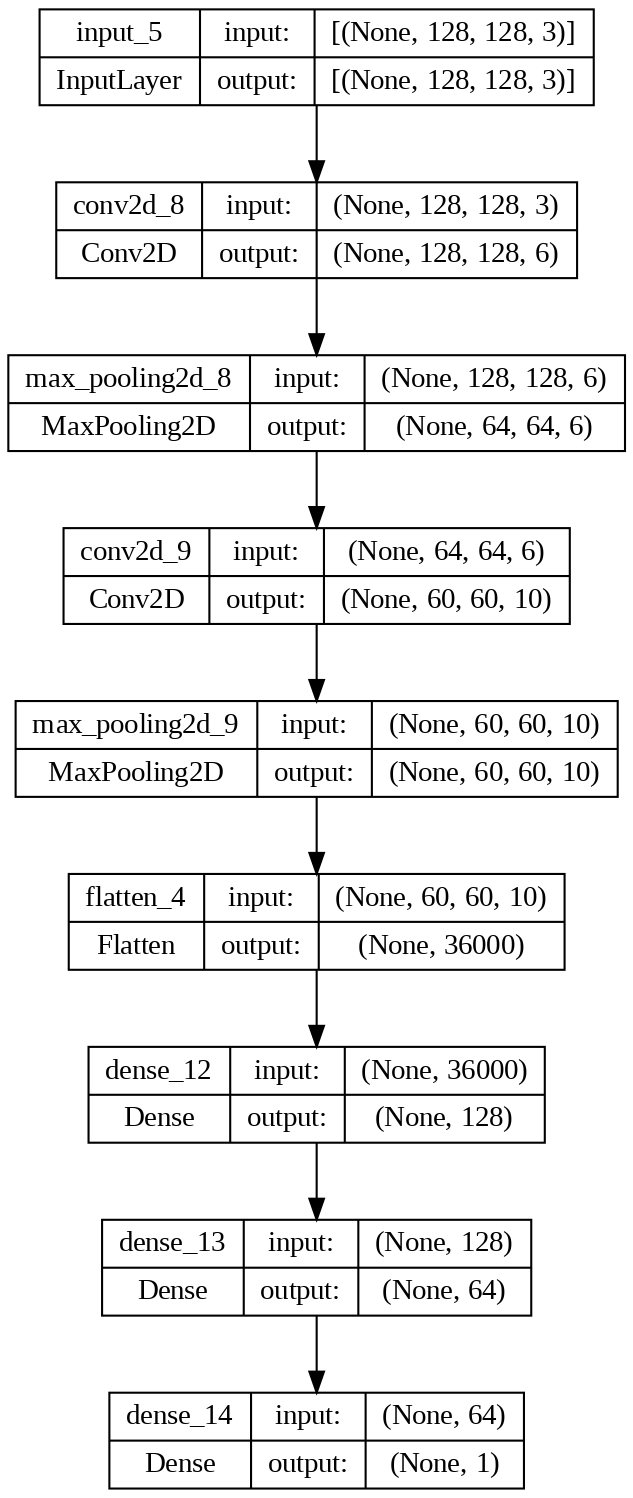


Figure 24 - Second model's dot format architecture

model\_2 = Sequential([

    Conv2D(6 , kernel\_size=5 , activation = "relu" , padding='same'),

    MaxPool2D(3 , strides=2 ,padding='same'),

    Conv2D(10 , kernel\_size=5 , activation='relu' , padding='valid'),

    MaxPool2D(3, strides=1, padding='same'),

    Flatten(),

    Dense(128 , activation='relu'),

    Dense(64 , activation='relu'),

    Dense(1, activation='sigmoid')

])

Then the model should be compiled with Adam as its optimizer and binary cross entropy as its loss function.

model\_2.compile(optimizer='adam',

              loss=

              'binary\_crossentropy',

              metrics=[tfa.metrics.F1Score(num\_classes=1), 'accuracy'])

The model is then trained with a batch size equal to 32 and for 10 epochs. Although we want to run it for 10 epochs, we set an early stopping condition which may terminate the training phase sooner than expected to eschew overfitting.

The best model is saved as the name of '2CNN-2FC\_Simple.h5' and can be seen in figure 25.

save\_dir = "./results/"

model\_name = '2CNN-2FC\_Simple.h5'

model\_path = os.path.join(save\_dir, model\_name)

history\_simple\_CNN2 = model\_2.fit(X\_train\_arr, y\_train\_arr,

          batch\_size=32,

          epochs=10,

          validation\_data=(X\_dev\_arr, y\_dev\_arr),

          callbacks=[

            ModelCheckpoint(model\_path, save\_best\_only=True),

            EarlyStopping(monitor='val\_loss', mode='min', patience=2, min\_delta=0.0001, verbose=2)

          ])



Figure 25 - Second model's output

And the training stage is shown below:

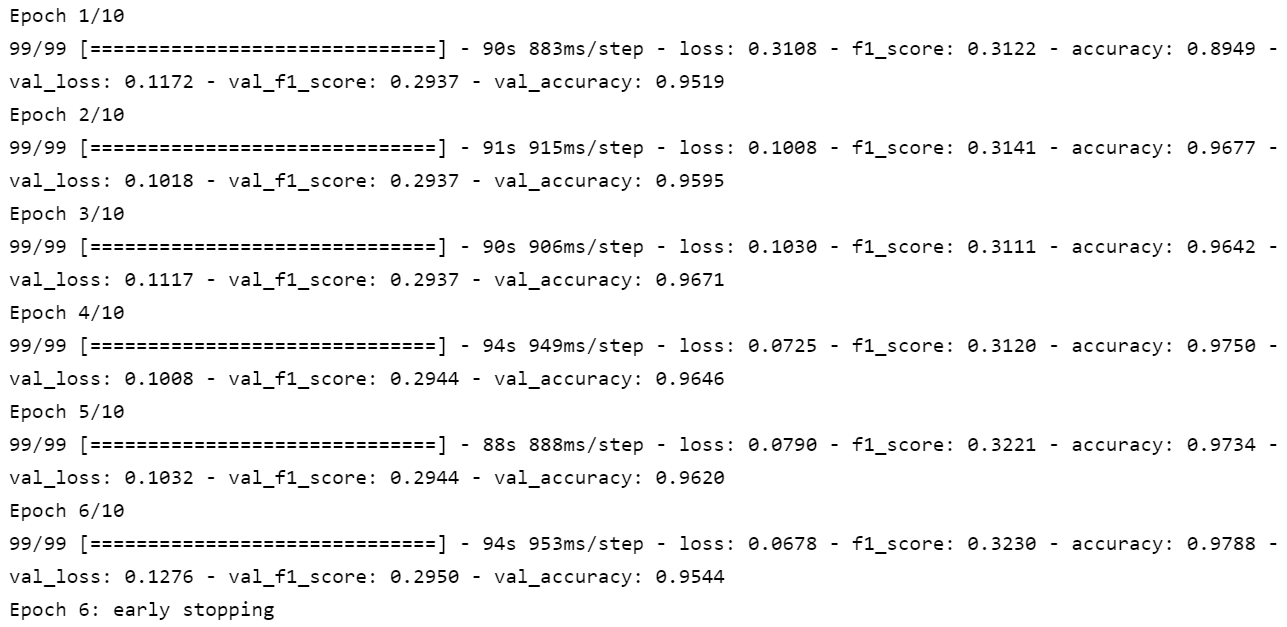


Figure 26 - Training stage for the second model

In the summary section, it can be seen that the model has 19,691,695 parameters all of which are trainable.

Additionally, the performance of the model on the train set and dev set according to each epoch is depicted below:

# plotting the metrics

fig = plt.figure(figsize=(13,6))

ax = plt.subplot(111)

ax.plot(history\_simple\_CNN2.history['loss'])

ax.plot(history\_simple\_CNN2.history['val\_loss'])

ax.set\_title('2CNN-2FC\_Simple loss')

ax.set\_ylabel('loss')

ax.set\_xlabel('epoch')

ax.legend(['train', 'test'], loc='upper right')

ax.grid(True)

plt.show()

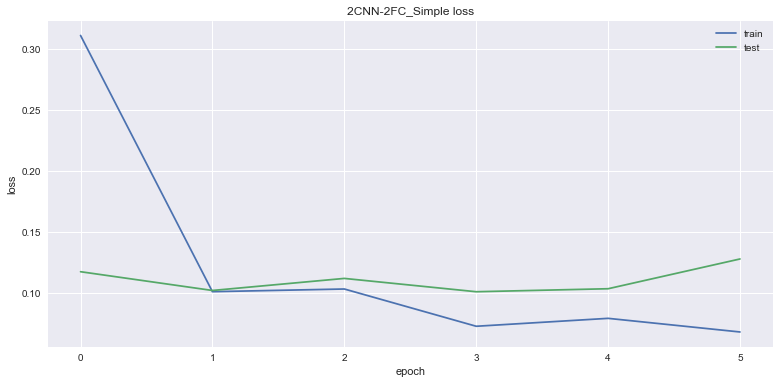


Figure 27 - Train and dev set loss for each epoch

#### 9.1.2.2. Model Assessment

We should assess the output of this model with the help of some measurements like accuracy, F1-score & etc., and also do some error analysis. Although we perform error analysis, we won’t change the structure of the model or augment data or anything else.

prediction = (model\_2.predict(X\_train\_arr)).squeeze()

prediction = np.where(prediction>=0.5 , 1 , 0)

print(classification\_report(prediction, y\_train\_arr))

print(confusion\_matrix(prediction, y\_train\_arr))

And the same codes are applied to dev & test sets, the results are as below



Figure 28 - Measurements for the second model on the train set

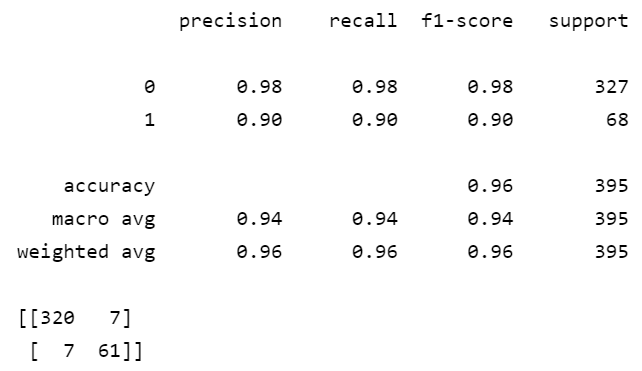


Figure 29 - Measurements for the second model on the dev set

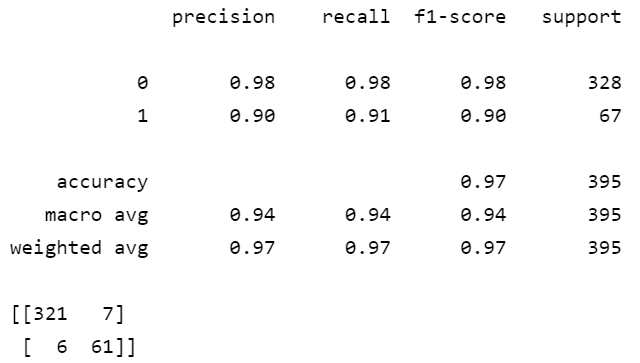


Figure 30 - Measurements for the second model on the test set

Because humans can perfectly detect whether someone has worn a mask or not, human-level performance error is equal to 0.00%.

Table 2 - Error analysis for the second model

|  |  |
| --- | --- |
| Type | Error |
| Human-level performance | 0.00% |
| Train set | 2% |
| Dev set | 4% |
| Test set | 3% |

As we terminated the model sooner than training to its full potential, it’s conspicuous that the error of the test set is low. We have encountered the problem of bias (if want to be rigid) because the train set error is not near human-level performance error. Nevertheless, it is negligible. So far, the best model is this one.

Surprisingly, even though we’ve applied a shallower network, the result is better.

### 9.1.3. Third Model

#### 9.1.3.1. Model Implementation

Our third model contains 2 convolutional layers (with max-pooling layers following them) and they are thicker and have more channels, and only one fully connected layer is accompanying them. The general architecture is as below:

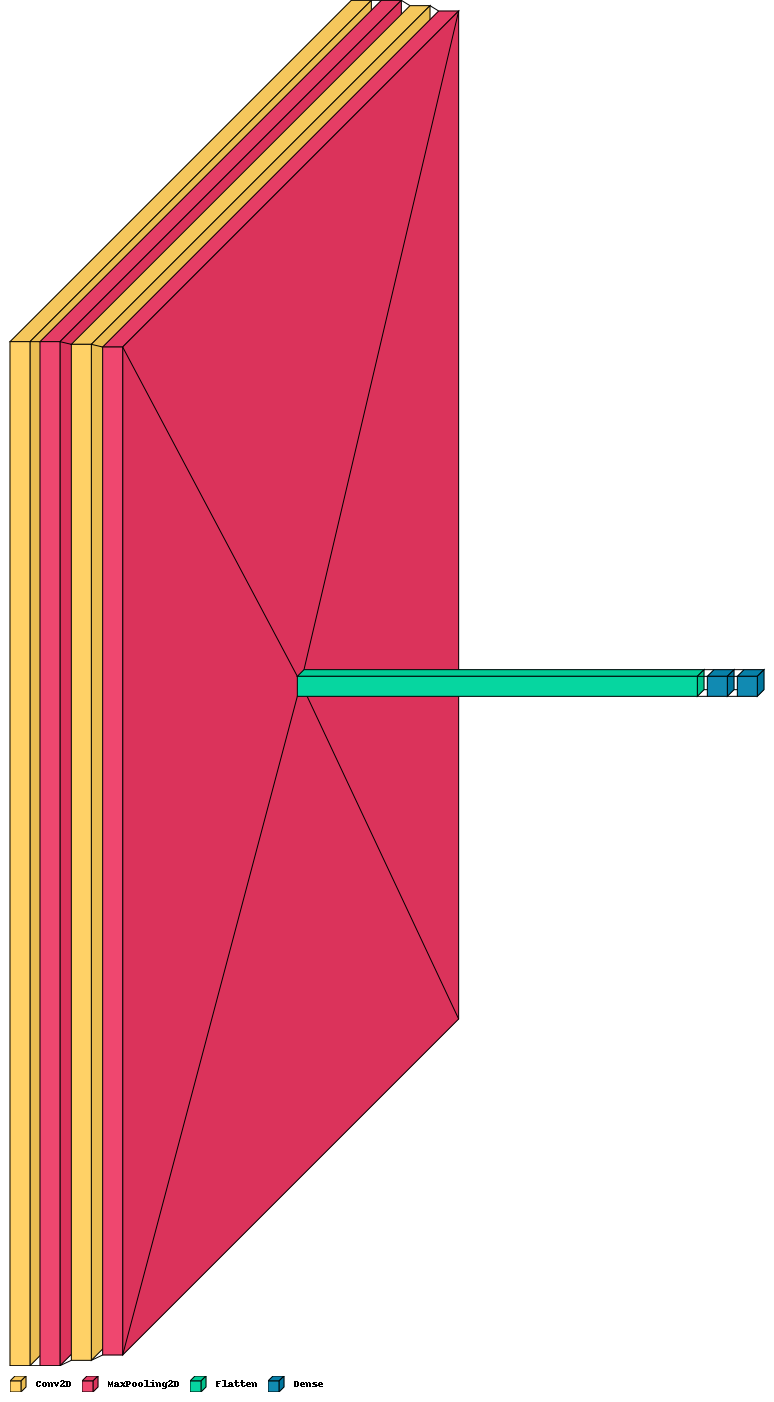


Figure 31 – Third model's architecture

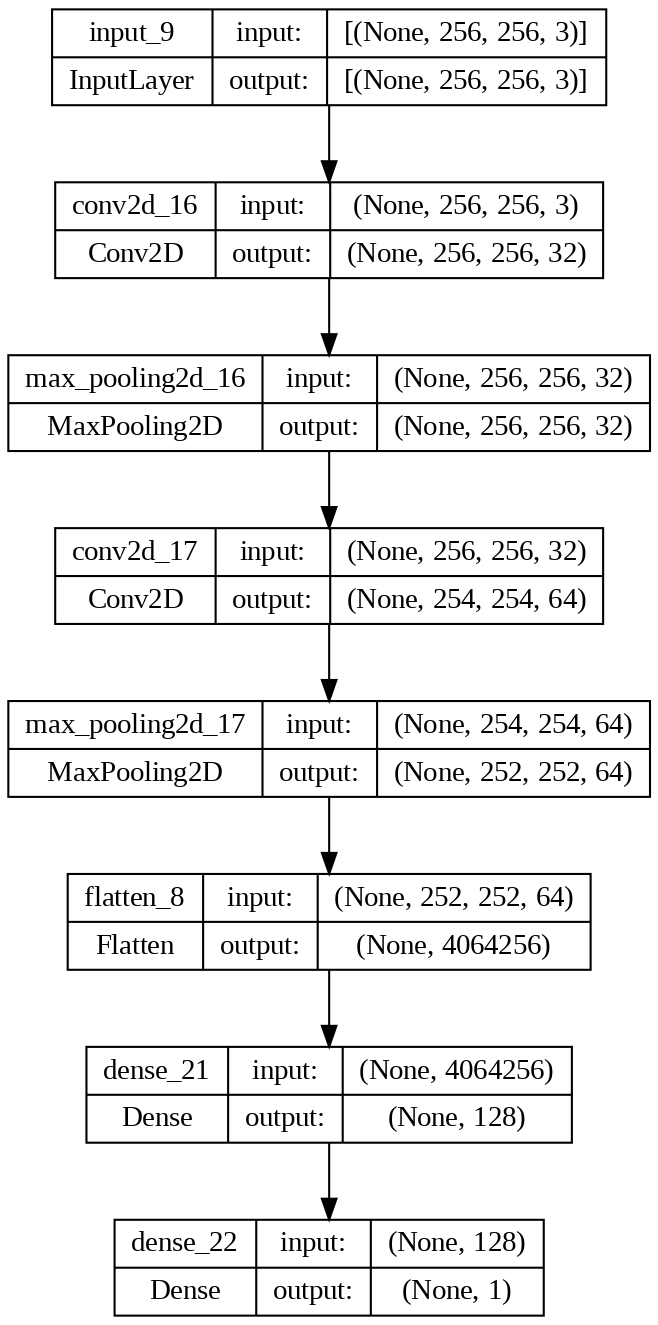


Figure 32 - Third model's dot format architecture

model\_3 = Sequential([

    Conv2D(32 , kernel\_size=3 , activation = "relu" , padding='same'),

    MaxPool2D(3, strides=1, padding='same'),

    Conv2D(64 , kernel\_size=3 , activation='relu' , padding='valid'),

    MaxPool2D(3, strides=1, padding='valid'),

    Flatten(),

    Dense(128 , activation='relu'),

    Dense(1, activation='sigmoid')

])

Then the model should be compiled with Adam as its optimizer and binary cross entropy as its loss function.

model\_3.compile(optimizer='adam',

              loss=

              'binary\_crossentropy',

              metrics=[tfa.metrics.F1Score(num\_classes=1), 'accuracy'])

The model is then trained with a batch size equal to 32 and for 10 epochs. Although we want to run it for 10 epochs, we set an early stopping condition which may terminate the training phase sooner than expected to eschew overfitting.

The best model is saved as the name of '2CNN-1FC\_Simple.h5' and can be seen in figure 33.

save\_dir = "./results/"

model\_name = '2CNN-1FC\_Simple.h5'

model\_path = os.path.join(save\_dir, model\_name)

history\_simple\_CNN3 = model\_3.fit(X\_train\_arr, y\_train\_arr,

          batch\_size=32,

          epochs=5,

          validation\_data=(X\_dev\_arr, y\_dev\_arr),

          callbacks=[

            ModelCheckpoint(model\_path, save\_best\_only=True),

            EarlyStopping(monitor='val\_loss', mode='min', patience=1, min\_delta=0.0001)

          ])



Figure 33 - Second model's output

And the training stage is shown below:

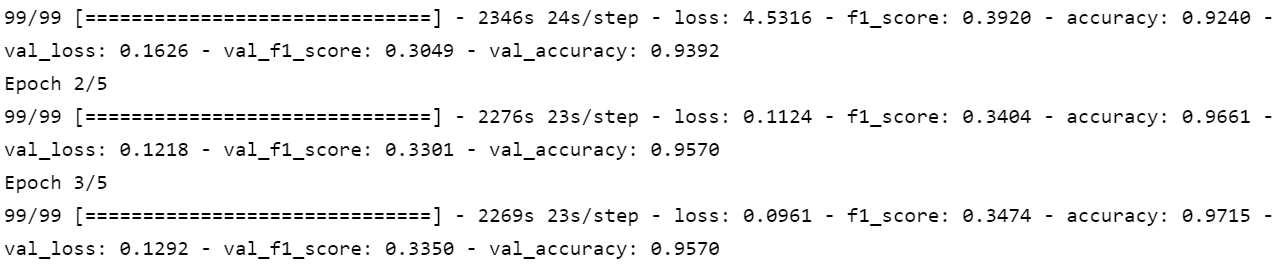


Figure 34 - Training stage for the second model

In the summary section, it can be seen that the model has 520,244,417 parameters all of which are trainable.

Additionally, the performance of the model on the train set and dev set according to each epoch is depicted below:

# plotting the metrics

fig = plt.figure(figsize=(13,6))

ax = plt.subplot(111)

ax.plot(history\_simple\_CNN3.history['loss'])

ax.plot(history\_simple\_CNN3.history['val\_loss'])

ax.set\_title('2CNN-FC\_Simple loss')

ax.set\_ylabel('loss')

ax.set\_xlabel('epoch')

ax.legend(['train', 'test'], loc='upper right')

ax.grid(True)

plt.show()

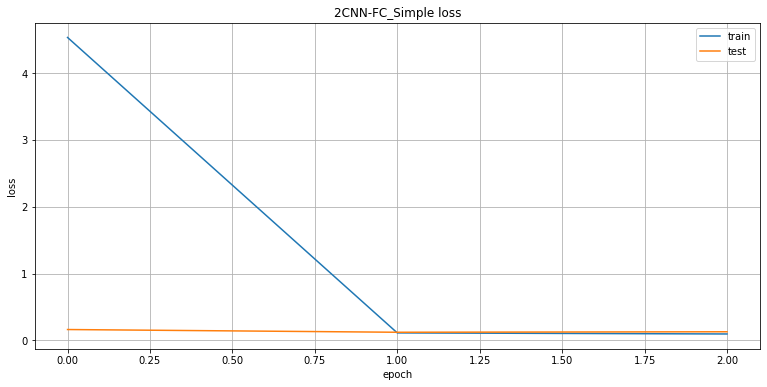


Figure 35 - Train and dev set loss for each epoch

#### 9.1.2.2. Model Assessment

We should assess the output of this model with the help of some measurements like accuracy, F1-score & etc., and also do some error analysis. Although we perform error analysis, we won’t change the structure of the model or augment data or anything else.

prediction = (model\_3.predict(X\_train\_arr)).squeeze()

prediction = np.where(prediction>=0.5 , 1 , 0)

print(classification\_report(prediction, y\_train\_arr))

print(confusion\_matrix(prediction, y\_train\_arr))

And the same codes are applied to dev & test sets, the results are as below:

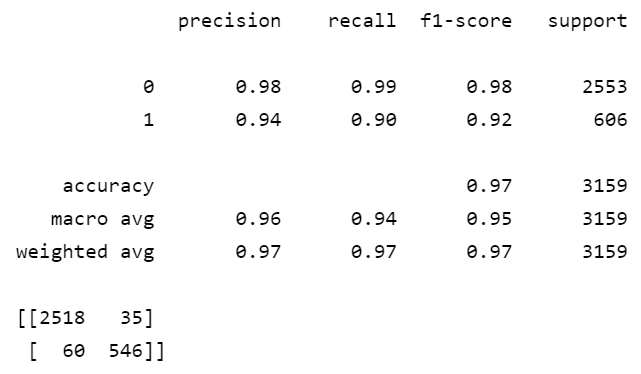


Figure 36 - Measurements for the third model on the train set



Figure 37 - Measurements for the third model on the dev set

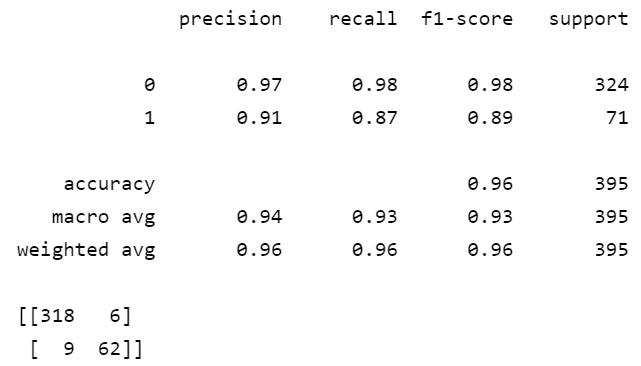


Figure 38 - Measurements for the third model on the test set

Because humans can perfectly detect whether someone has worn a mask or not, human-level performance error is equal to 0.00%.

Table 3 - Error analysis for the second model

|  |  |
| --- | --- |
| Type | Error |
| Human-level performance | 0.00% |
| Train set | 3% |
| Dev set | 4% |
| Test set | 4% |

The problem of bias still exists and the options given for the first model can be applied. Another interesting point is that, even though we kind of achieved the same amount of accuracy with this model and with the first model, the first model is preferred. Because its training was much less computationally expensive. This model took about 2 hours to run only 2 epochs.

## 9.2. Transfer Learning

In this section, we are after implementing some networks and getting help from transfer learning. Because we have less than 10000 images, we won’t tune the base models’ weights (of the models which were used) and only add some fully connected layers afterward and train the model on those final layers. These models were trained on [ImageNet](https://www.image-net.org/) dataset.

Also, consider that in this section we omitted the dev set and will not perform error analysis and only report the final error on train and test sets.

At last, the codes of this section (as it was mentioned before) are implemented in google colab.

### 9.2.1. MobileNetV2

#### 9.2.1.1. Model Implementation

First, we have to preprocess the images based on the characteristics of this specific model.

X\_train\_arr, X\_test\_arr = tuple([tf.keras.applications.mobilenet\_v2.preprocess\_input(x) for x in [X\_train\_arr, X\_test\_arr]])

After that we import the MobileNetV2 and truncate its head, we add some fully connected layers on top.

baseModel = tf.keras.applications.MobileNetV2(weights="imagenet", include\_top=False,

  input\_tensor = tfl.Input(shape=(224, 224, 3)))

# construct the head of the model that will be placed on top of the

# the base model

headModel = baseModel.output

# headModel = tfl.MaxPooling2D(pool\_size=(7, 7))(headModel)

headModel = Flatten(name="flatten")(headModel)

headModel = Dense(256, activation="relu")(headModel)

headModel = tfl.Dropout(0.4)(headModel)

headModel = Dense(128, activation="relu")(headModel)

headModel = tfl.Dropout(0.3)(headModel)

headModel = Dense(64, activation="relu")(headModel)

headModel = tfl.Dropout(0.1)(headModel)

headModel = Dense(1, activation="sigmoid")(headModel)

# place the head FC model on top of the base model (this will become

# the actual model we will train)

model = tf.keras.models.Model(inputs=baseModel.input, outputs=headModel)

# loop over all layers in the base model and freeze them so they will

# \*not\* be updated during the first training process

for layer in baseModel.layers:

  layer.trainable = False

The overall look of the model is delineated below:

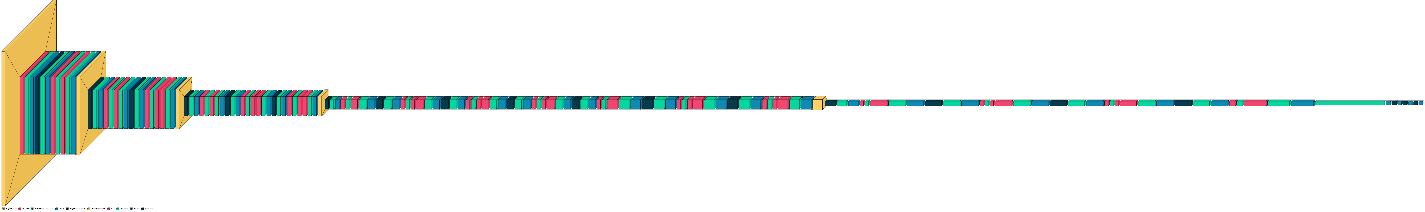


Figure 39 - MobileNetV2 architecture

Then the model should be compiled with Adam as its optimizer and binary cross entropy as its loss function.

model.compile(optimizer='adam',

              loss=

              'binary\_crossentropy',

              metrics=[tfa.metrics.F1Score(num\_classes=1), 'accuracy'])

The model is then trained with a batch size equal to 1 and for 50 epochs. Although we want to run it for 50 epochs, we set an early stopping condition which may terminate the training phase sooner than expected to eschew overfitting.

The best model is saved as the name of 'MobileNetV2-edited.h5' and can be seen in figure 40.

save\_dir = "./results/"

model\_name = 'MobileNetV2-edited.h5'

model\_path = os.path.join(save\_dir, model\_name)

history\_cmplex\_MovileNetV2 = model.fit(X\_train\_arr, y\_train\_arr,

          batch\_size=1,

          epochs=50,

          validation\_data=(X\_test\_arr, y\_test\_arr),

          callbacks=[

            ModelCheckpoint(model\_path, save\_best\_only=True),

            EarlyStopping(monitor='val\_loss', mode='min', patience=3, min\_delta=0.0001)

          ])

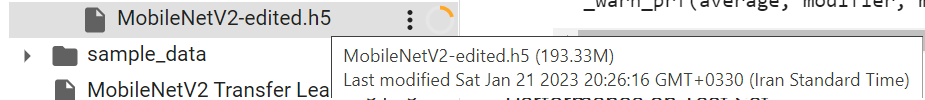


Figure 40 – MobileNetV2 output

And the training stage is shown below:

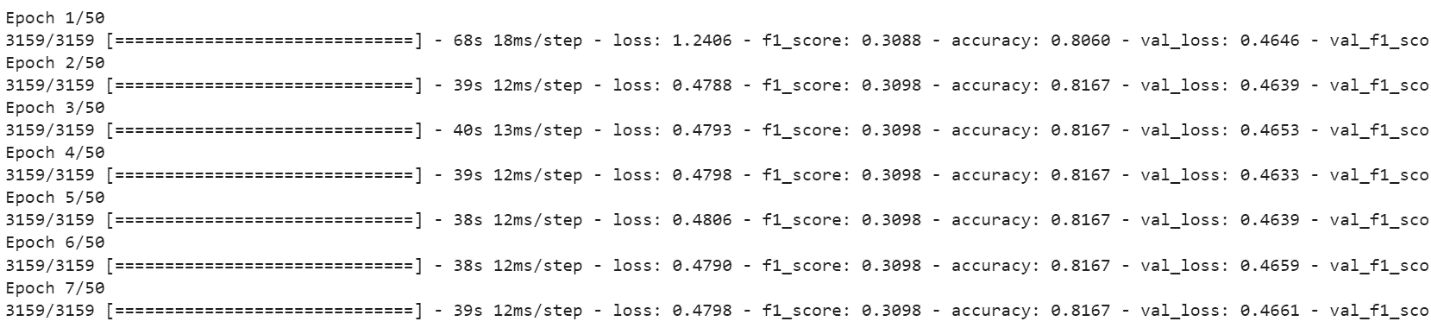


Figure 41 - Training stage for the MobileNetV2 model

In the summary section, it can be seen that the model has 18,355,777 parameters and 16,097,793 are trainable.

Additionally, the performance of the model on the train set and dev set according to each epoch is depicted below:

# plotting the metrics

fig = plt.figure(figsize=(13,6))

plt.style.use('seaborn')

ax = plt.subplot(111)

ax.plot(history\_cmplex\_MovileNetV2.history['loss'])

ax.plot(history\_cmplex\_MovileNetV2.history['val\_loss'])

ax.set\_title('MobileNetV2 Transfer Learning')

ax.set\_ylabel('loss')

ax.set\_xlabel('epoch')

ax.legend(['train', 'test'], loc='upper right')

ax.grid(True)

plt.show()

plt.savefig('MobileNetV2 Transfer Learning.png',dpi=200)

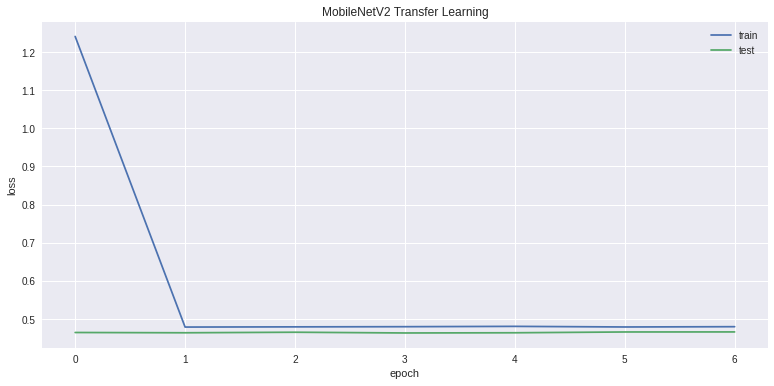


Figure 42 - Train and test set loss for each epoch

#### 9.2.1.2. Model Assessment

The errors of training and test sets are calculated below:

prediction = (model.predict(X\_train\_arr)).squeeze()

prediction = np.where(prediction>=0.5 , 1 , 0)

print(classification\_report(prediction, y\_train\_arr))

print(confusion\_matrix(prediction, y\_train\_arr))

And the same snippet of code is applied to the test set, the results are as below

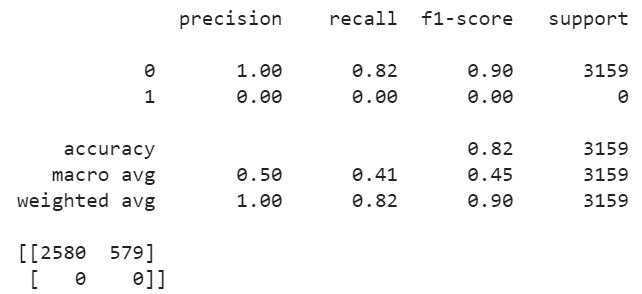


Figure 43 - Measurements for MobileNetV2 on train set

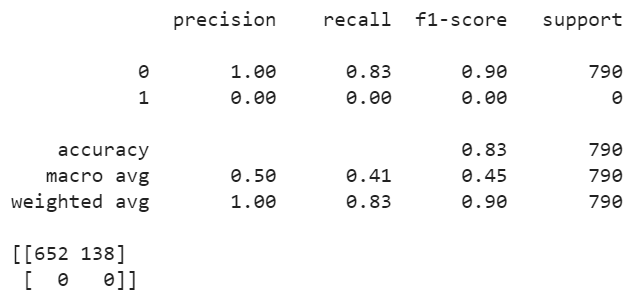


Figure 44 - Measurements for MobileNetV2 on the test set

This model works poorly both on training and test sets because it doesn’t recognize any of the images without a mask and is not different from a dummy classifier which predicts the label of the class which is more frequent.

### 9.2.2. VGG-19

#### 9.2.2.1. Model Implementation

First, we have to preprocess the images based on the characteristics of this specific model.

X\_train\_arr\_vgg, X\_test\_arr\_vgg = tuple([tf.keras.applications.vgg19.preprocess\_input(x) for x in [X\_train\_arr, X\_test\_arr]])

After that, we import the VGG-19 and truncate its head, and add some fully connected layers on top.

baseModel = VGG19(weights="imagenet", include\_top=False,

  input\_tensor = tfl.Input(shape=(224, 224, 3)))

# construct the head of the model that will be placed on top of the

# the base model

headModel = baseModel.output

headModel = tfl.MaxPooling2D(pool\_size=(3, 3))(headModel)

headModel = Flatten(name="flatten")(headModel)

headModel = Dense(128, activation="relu")(headModel)

headModel = tfl.Dropout(0.1)(headModel)

headModel = Dense(64, activation="relu")(headModel)

headModel = Dense(1, activation="sigmoid")(headModel)

# place the head FC model on top of the base model (this will become

# the actual model we will train)

model\_vgg = tf.keras.models.Model(inputs=baseModel.input, outputs=headModel)

# loop over all layers in the base model and freeze them so they will

# \*not\* be updated during the first training process

for layer in baseModel.layers:

  layer.trainable = False

The overall look of the model is delineated below:

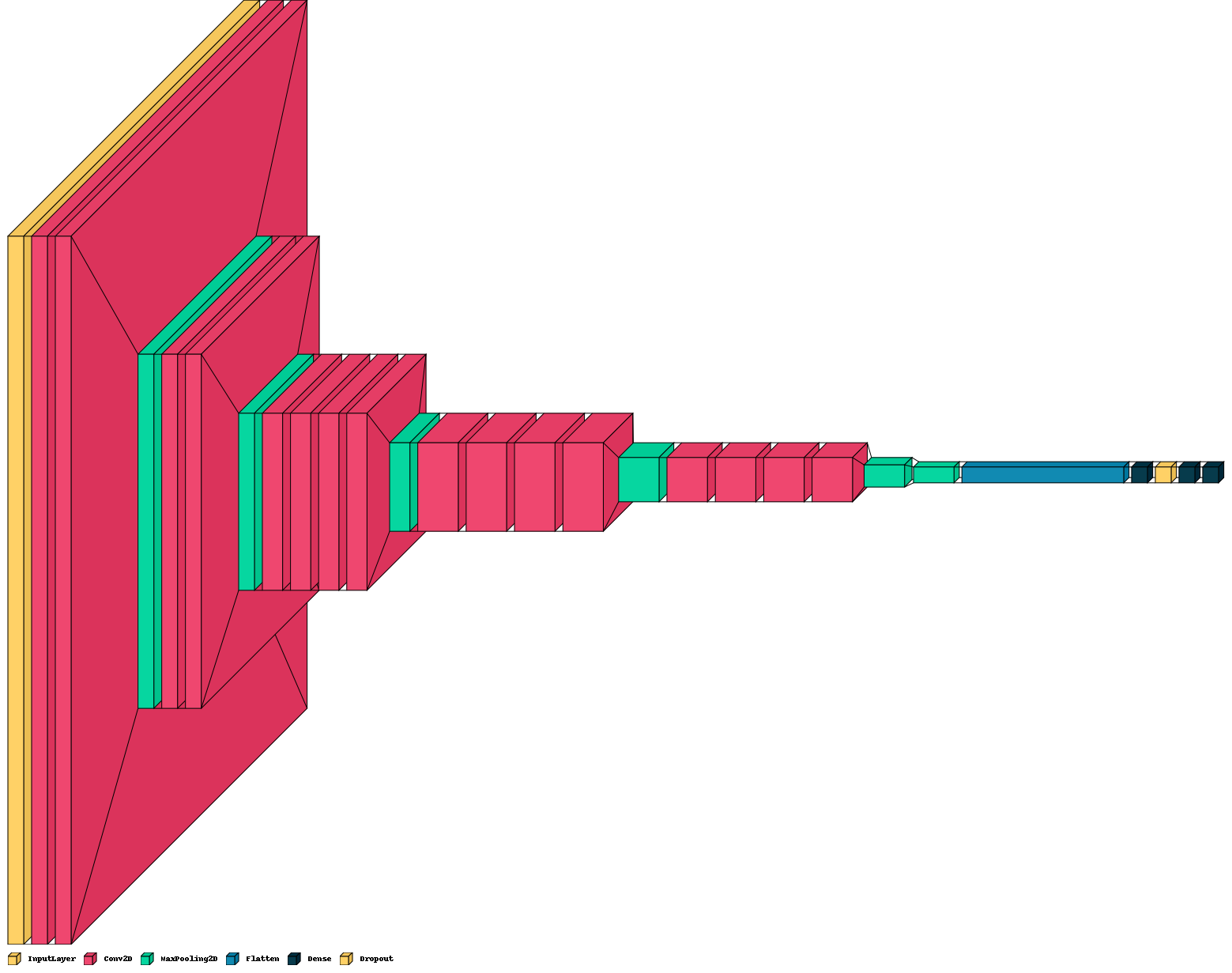


Figure 45 – VGG-19 architecture

Then the model should be compiled with Adam as its optimizer and binary cross entropy as its loss function.

model\_vgg.compile(optimizer='adam',

              loss=

              'binary\_crossentropy',

              metrics=[tfa.metrics.F1Score(num\_classes=1), 'accuracy'])

The model is then trained with a batch size equal to 1 and for 10 epochs. Although we want to run it for 10 epochs, we set an early stopping condition which may terminate the training phase sooner than expected to eschew overfitting.

The best model is saved as the name of 'VGG19-edited.h5' and can be seen in figure 46.

save\_dir = "./results/"

model\_name = 'VGG19-edited.h5'

model\_path = os.path.join(save\_dir, model\_name)

history\_cmplex\_VGG19 = model\_vgg.fit(X\_train\_arr\_vgg, y\_train\_arr,

          batch\_size=1,

          epochs=10,

          validation\_data=(X\_test\_arr\_vgg, y\_test\_arr),

          callbacks=[

            ModelCheckpoint(model\_path, save\_best\_only=True),

            EarlyStopping(monitor='val\_loss', mode='min', patience=3, min\_delta=0.0001)

          ])

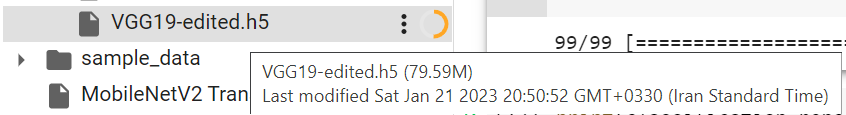


Figure 46 - VGG-19 output

And the training stage is shown below:

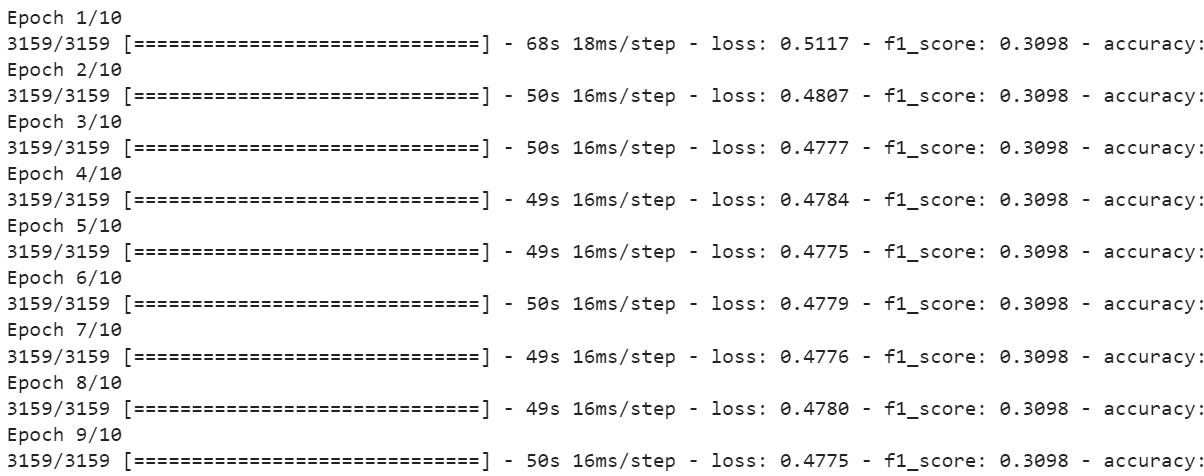


Figure 47 - Training stage for the VGG-19 model

In the summary section, it can be seen that the model has 20,294,977 parameters and 270,593 are trainable.

Additionally, the performance of the model on the train set and dev set according to each epoch is depicted below:

# plotting the metrics

fig = plt.figure(figsize=(13,6))

ax = plt.subplot(111)

ax.plot(history\_cmplex\_VGG19.history['loss'])

ax.plot(history\_cmplex\_VGG19.history['val\_loss'])

ax.set\_title('VGG19 Transfer Learning)

ax.set\_ylabel('loss')

ax.set\_xlabel('epoch')

ax.legend(['train', 'test'], loc='upper right')

ax.grid(True)

plt.show()

plt.savefig('VGG19 Transfer Learning.png',dpi=200)

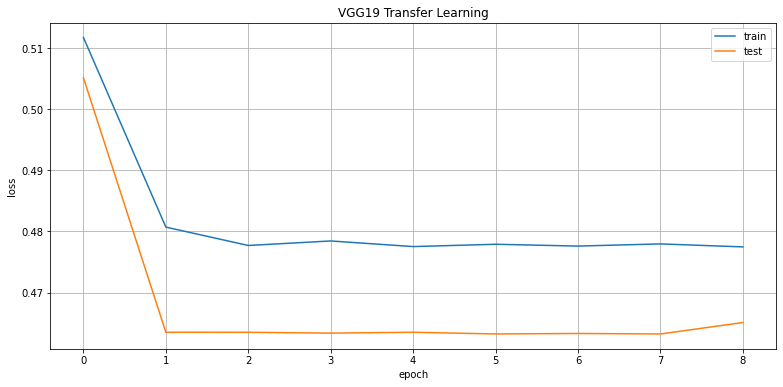


Figure 48 - Train and test set loss for each epoch

#### 9.2.2.2. Model Assessment

The errors of training and test sets are calculated below:

prediction = (model\_vgg.predict(X\_train\_arr\_vgg)).squeeze()

prediction = np.where(prediction>=0.5 , 1 , 0)

print(classification\_report(prediction, y\_train\_arr))

print(confusion\_matrix(prediction, y\_train\_arr))

And the same snippet of code is applied to the test set, the results are as below

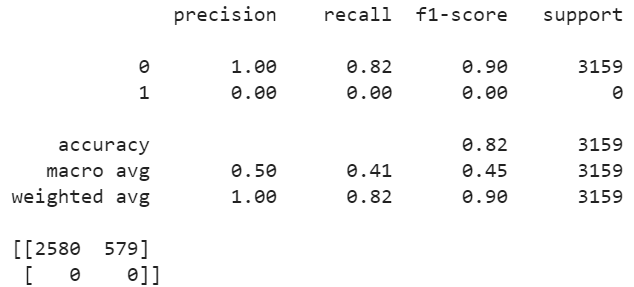


Figure 49 - Measurements for VGG-19 on train set

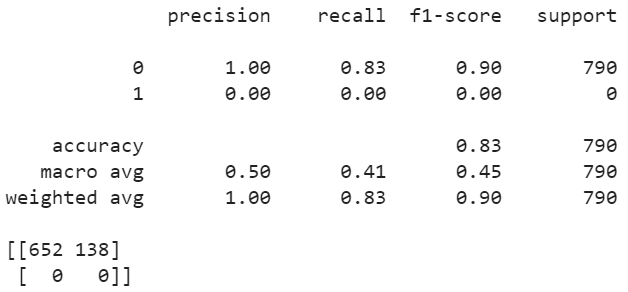


Figure 50 - Measurements for VGG-19 on the test set

This model works poorly both on training and test sets because it doesn’t recognize any of the images without a mask and is not different from a dummy classifier which predicts the label of the class which is more frequent.

### 9.2.3. ResNet-50

#### 9.2.3.1. Model Implementation

First, we have to preprocess the images based on the characteristics of this specific model.

X\_train\_arr\_res, X\_test\_arr\_res = tuple([tf.keras.applications.resnet50.preprocess\_input(x) for x in [X\_train\_arr, X\_test\_arr]])

After that we import the ResNet-50 and truncate its head, we add some fully connected layers on top.

baseModel = ResNet50(weights="imagenet", include\_top=False,

  input\_tensor = tfl.Input(shape=(224, 224, 3)))

# construct the head of the model that will be placed on top of the

# the base model

headModel = baseModel.output

headModel = tfl.MaxPooling2D(pool\_size=(3, 3))(headModel)

headModel = Flatten(name="flatten")(headModel)

headModel = Dense(128, activation="relu")(headModel)

headModel = tfl.Dropout(0.1)(headModel)

headModel = Dense(64, activation="relu")(headModel)

headModel = Dense(1, activation="sigmoid")(headModel)

# place the head FC model on top of the base model (this will become

# the actual model we will train)

model\_resnet = tf.keras.models.Model(inputs=baseModel.input, outputs=headModel)

# loop over all layers in the base model and freeze them so they will

# \*not\* be updated during the first training process

for layer in baseModel.layers:

  layer.trainable = False

The overall look of the model is delineated below:



Figure 51 – ResNet-50 architecture

Then the model should be compiled with Adam as its optimizer and binary cross entropy as its loss function.

model\_resnet.compile(optimizer='adam',

              loss=

              'binary\_crossentropy',

              metrics=[tfa.metrics.F1Score(num\_classes=1), 'accuracy'])

The model is then trained with a batch size equal to 1 for 50 epochs. Although we want to run it for 50 epochs, we set an early stopping condition which may terminate the training phase sooner than expected to eschew overfitting.

The best model is saved as the name of 'ResNet50-edited.h5'and can be seen in figure 52.

save\_dir = "./results/"

model\_name = 'ResNet50-edited.h5'

model\_path = os.path.join(save\_dir, model\_name)

history\_cmplex\_ResNet50 = model\_resnet.fit(X\_train\_arr\_res, y\_train\_arr,

          batch\_size=1,

          epochs=50,

          validation\_data=(X\_test\_arr\_res, y\_test\_arr),

          callbacks=[

            ModelCheckpoint(model\_path, save\_best\_only=True),

            EarlyStopping(monitor='val\_loss', mode='min', patience=3, min\_delta=0.0001)

          ])

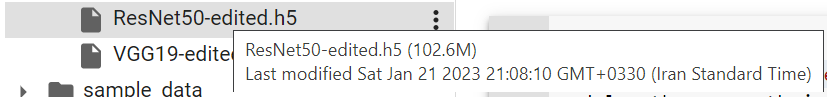


Figure 52 – ResNet-50 output

And the training stage is shown below:

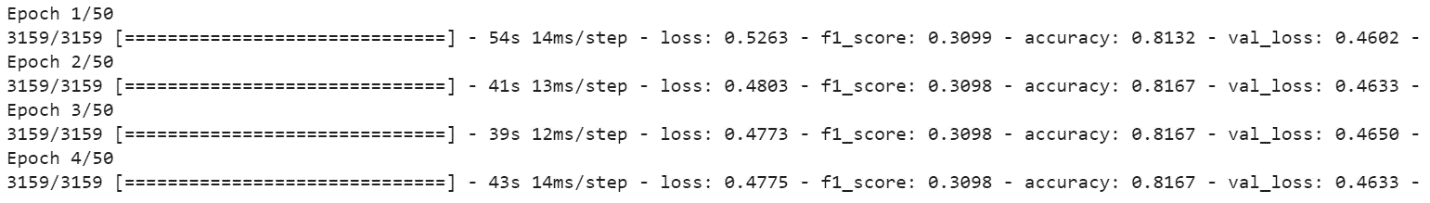


Figure 53 - Training stage for the ResNet-50 model

In the summary section, it can be seen that the model has 24,644,737 parameters and 1,057,025 are trainable.

Additionally, the performance of the model on the train set and dev set according to each epoch is depicted below:

# plotting the metrics

fig = plt.figure(figsize=(13,6))

ax = plt.subplot(111)

ax.plot(history\_cmplex\_ResNet50.history['loss'])

ax.plot(history\_cmplex\_ResNet50.history['val\_loss'])

ax.set\_title('ResNet50 Transfer Learning')

ax.set\_ylabel('loss')

ax.set\_xlabel('epoch')

ax.legend(['train', 'test'], loc='upper right')

ax.grid(True)

plt.show()

plt.savefig('ResNet50 Transfer Learning.png',dpi=200)

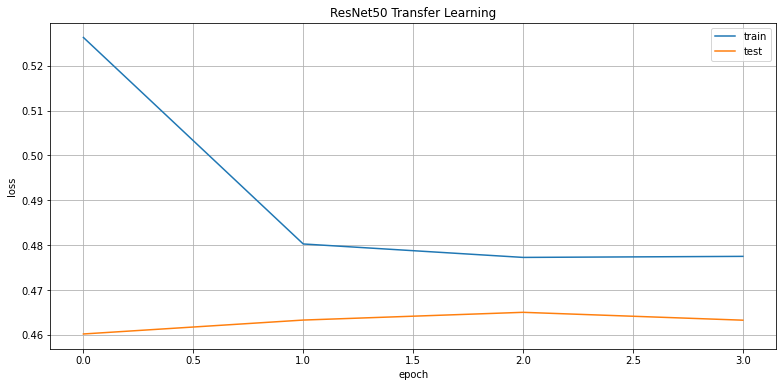


Figure 54 - Train and test set loss for each epoch

#### 9.2.3.2. Model Assessment

The errors of training and test sets are calculated below:

prediction = (model\_resnet.predict(X\_train\_arr\_res)).squeeze()

prediction = np.where(prediction>=0.5 , 1 , 0)

print(classification\_report(prediction, y\_train\_arr))

print(confusion\_matrix(prediction, y\_train\_arr))

And the same snippet of code is applied to the test set, the results are as below

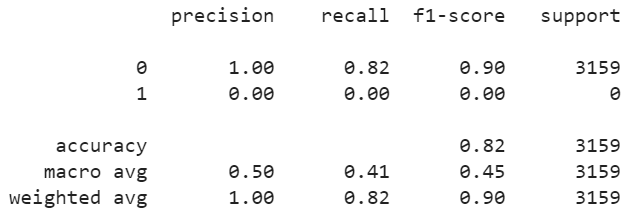


Figure 55 - Measurements for ResNet-50 on train set

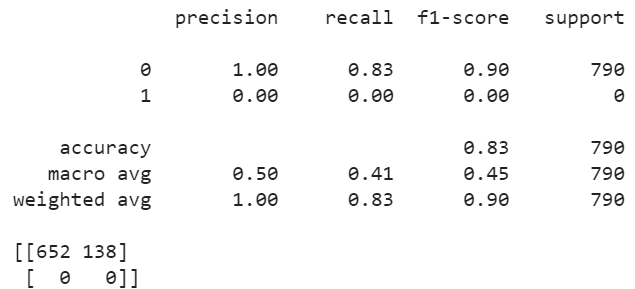


Figure 56 - Measurements for ResNet-50 on the test set

This model works poorly both on training and test sets because it doesn’t recognize any of the images without a mask and is not different from a dummy classifier which predicts the label of the class which is more frequent.

### 9.2.4. Inception-V3

#### 9.2.4.1. Model Implementation

First, we have to preprocess the images based on the characteristics of this specific model.

X\_train\_arr\_incep, X\_test\_arr\_incep = tuple([tf.keras.applications.inception\_v3.preprocess\_input(x) for x in [X\_train\_arr, X\_test\_arr]])

After that we import the Inception-V3 and truncate its head, we add some fully connected layers on top.

baseModel = InceptionV3(weights="imagenet", include\_top=False,

  input\_tensor = tfl.Input(shape=(224, 224, 3)))

# construct the head of the model that will be placed on top of the

# the base model

headModel = baseModel.output

headModel = tfl.MaxPooling2D(pool\_size=(3, 3))(headModel)

headModel = Flatten(name="flatten")(headModel)

headModel = Dense(128, activation="relu")(headModel)

headModel = tfl.Dropout(0.1)(headModel)

headModel = Dense(64, activation="relu")(headModel)

headModel = Dense(1, activation="sigmoid")(headModel)

# place the head FC model on top of the base model (this will become

# the actual model we will train)

model\_inception = tf.keras.models.Model(inputs=baseModel.input, outputs=headModel)

# loop over all layers in the base model and freeze them so they will

# \*not\* be updated during the first training process

for layer in baseModel.layers:

  layer.trainable = False

The overall look of the model is delineated below:



Figure 57 – Inception-V3 architecture

Then the model should be compiled with Adam as its optimizer and binary cross entropy as its loss function.

model\_inception.compile(optimizer='adam',

              loss=

              'binary\_crossentropy',

              metrics=[tfa.metrics.F1Score(num\_classes=1), 'accuracy'])

The model is then trained with a batch size equal to 1 and for 50 epochs. Although we want to run it for 50 epochs, we set an early stopping condition which may terminate the training phase sooner than expected to eschew overfitting.

The best model is saved as the name of 'InceptionV3-edited.h5' and can be seen in figure 58.

save\_dir = "./results/"

model\_name = 'InceptionV3-edited.h5'

model\_path = os.path.join(save\_dir, model\_name)

history\_cmplex\_InceptionV3 = model\_inception.fit(X\_train\_arr\_incep, y\_train\_arr,

          batch\_size=1,

          epochs=50,

          validation\_data=(X\_test\_arr\_incep, y\_test\_arr),

          callbacks=[

            ModelCheckpoint(model\_path, save\_best\_only=True),

            EarlyStopping(monitor='val\_loss', mode='min', patience=3, min\_delta=0.0001)

          ])

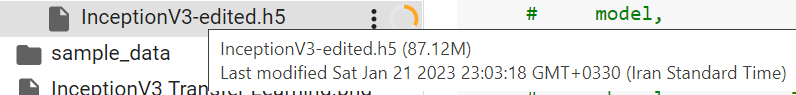


Figure 58 – Inception-V3 output

And the training stage is shown below:

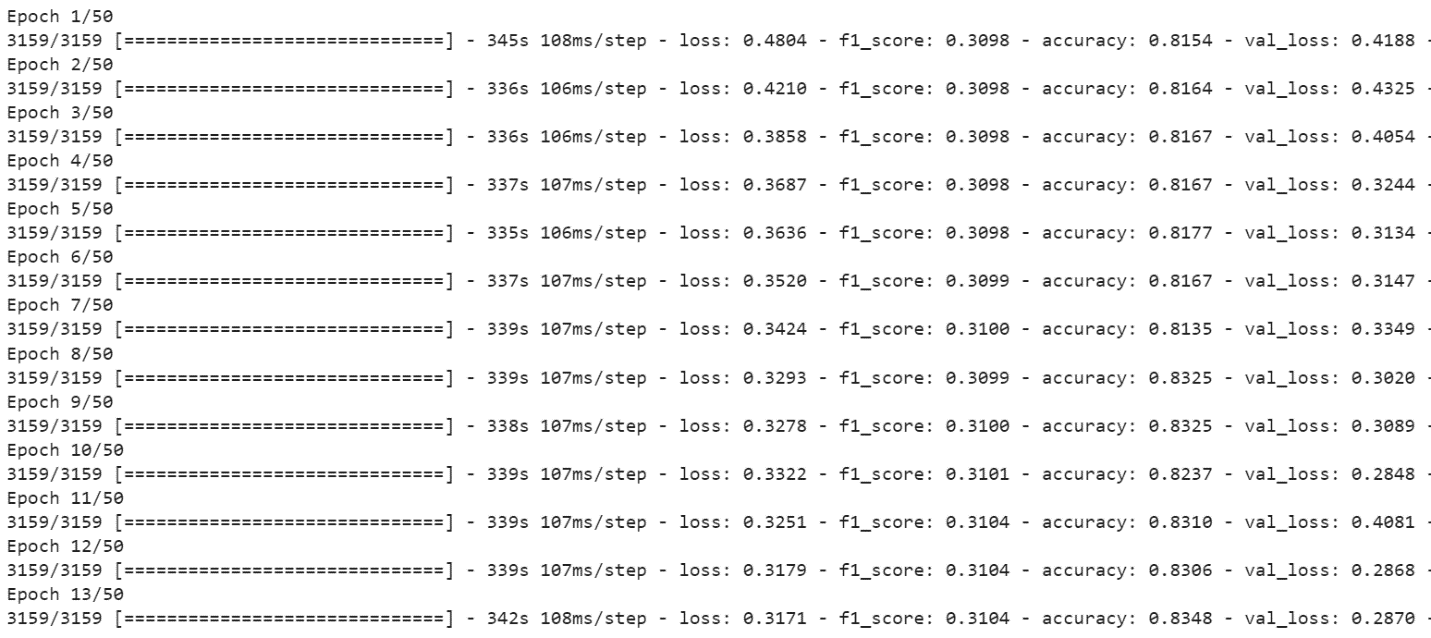


Figure 59 - Training stage for the Inception-V3 model

In the summary section, it can be seen that the model has 22,073,377 parameters and 270,593 are trainable.

Additionally, the performance of the model on the train set and dev set according to each epoch is depicted below:

# plotting the metrics

fig = plt.figure(figsize=(13,6))

ax = plt.subplot(111)

ax.plot(history\_cmplex\_InceptionV3.history['loss'])

ax.plot(history\_cmplex\_InceptionV3.history['val\_loss'])

ax.set\_title('InceptionV3 Transfer Learning')

ax.set\_ylabel('loss')

ax.set\_xlabel('epoch')

ax.legend(['train', 'test'], loc='upper right')

ax.grid(True)

plt.show()

plt.savefig('InceptionV3 Transfer Learning.png',dpi=200)

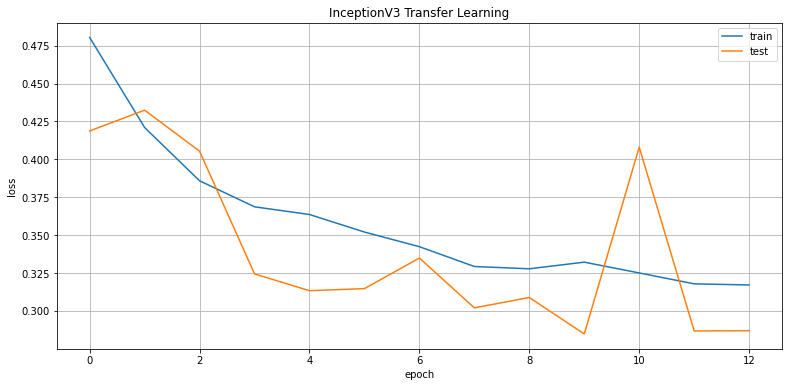


Figure 60 - Train and test set loss for each epoch

#### 9.2.4.2. Model Assessment

The errors of training and test sets are calculated below:

prediction = (model\_inception.predict(X\_train\_arr\_incep)).squeeze()

prediction = np.where(prediction>=0.5 , 1 , 0)

print(classification\_report(prediction, y\_train\_arr))

print(confusion\_matrix(prediction, y\_train\_arr))

And the same snippet of code is applied to the test set, the results are as below.

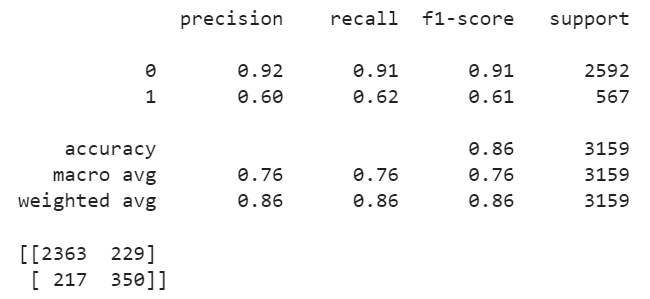


Figure 61 - Measurements for Inception-V3 on train set

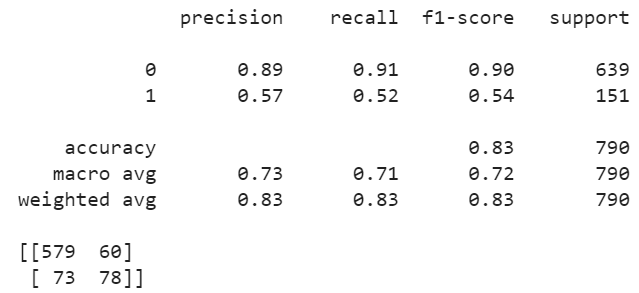


Figure 62 - Measurements for Inception-V3 on the test set

It can be seen that this model has a better performance on both training and test sets, since it at least predicts some images’ label as class one. Thus, among the base models, Inception-V3 possesses the best performance.

# 10. Misclassified Images

Best model so far is the second simple model implemented in section 9.1.2. having the accuracy of 97% on the test set. Thus, we first import the model and predict the test set with it.

model = tf.keras.models.load\_model('results/2CNN-2FC\_Simple.h5')

prediction = (model.predict(X\_test\_arr)).squeeze()

prediction = np.where(prediction>=0.5 , 1 , 0)

And then filter the images that are misclassified:

df\_test = df.loc[X\_test.index]

df\_test['prediction'] = prediction

mask = df\_test['prediction'] != df\_test['image\_label']

df\_diff = df\_test.loc[mask]

df\_diff

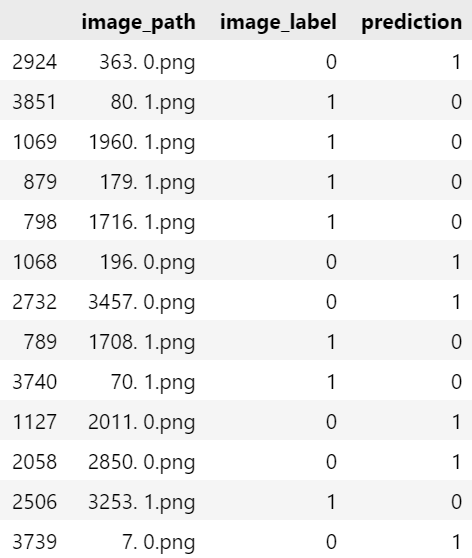


Figure 63 - Misclassified images based on the best model

Finally, we can observe the images below:

path= 'edited pics/'

dic = {0: 'wearing mask' , 1:'not wearing mask'}

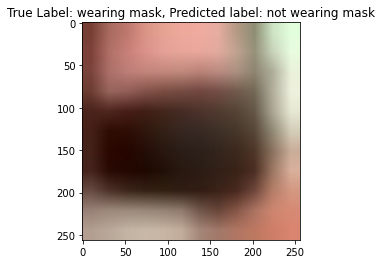
for idx in df\_diff.index:

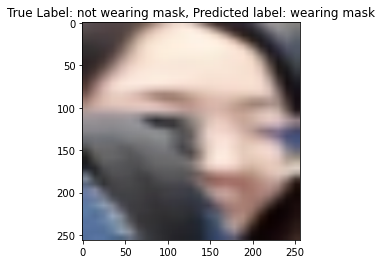
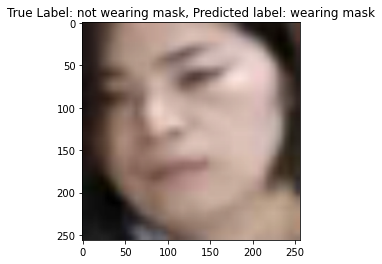
    plt.title(f"True Label: {dic[df\_diff.loc[idx , 'image\_label']]}, Predicted label: {dic[df\_diff.loc[idx , 'prediction']]}")

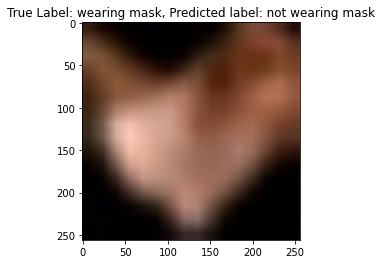
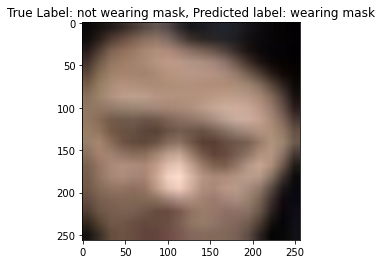
    image = mpimg.imread(path + df\_diff.loc[idx , 'image\_path'])

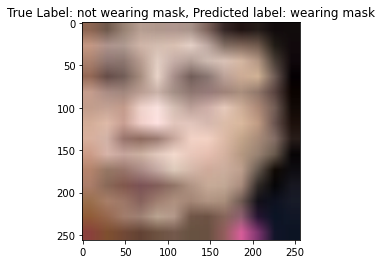
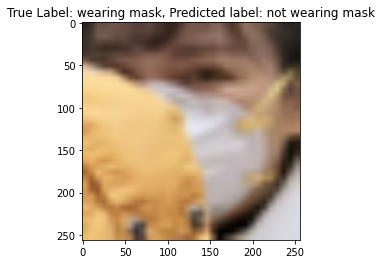
    plt.imshow(image)

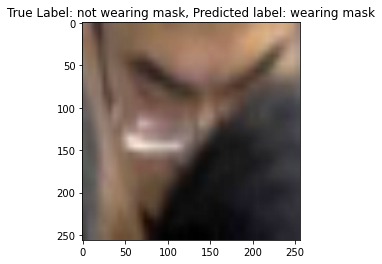
    plt.show()

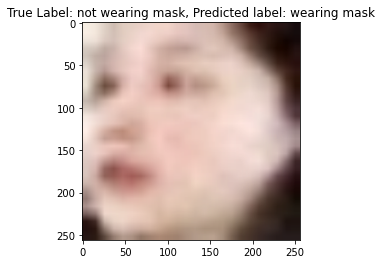


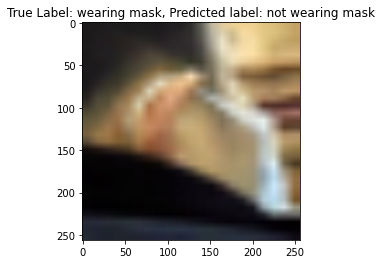












It can be deducted that except 6 images that their status is apparent, others are either low-resolution or not standing on the front. In general, the performance of the model is outstanding.