Amirkabir University of Technology

(Tehran Polytechnic)

Department of Industrial Engineering & Management Systems

Final Project

Mask Detection Using Convolutional Neural Networks

By:

Pedram Peiro Asfia - 9825006

Professor:

Dr. Marzieh Zarrinbal Masouleh

Course:

Artificial Intelligence

Jan 2023

Contents

Figures

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.

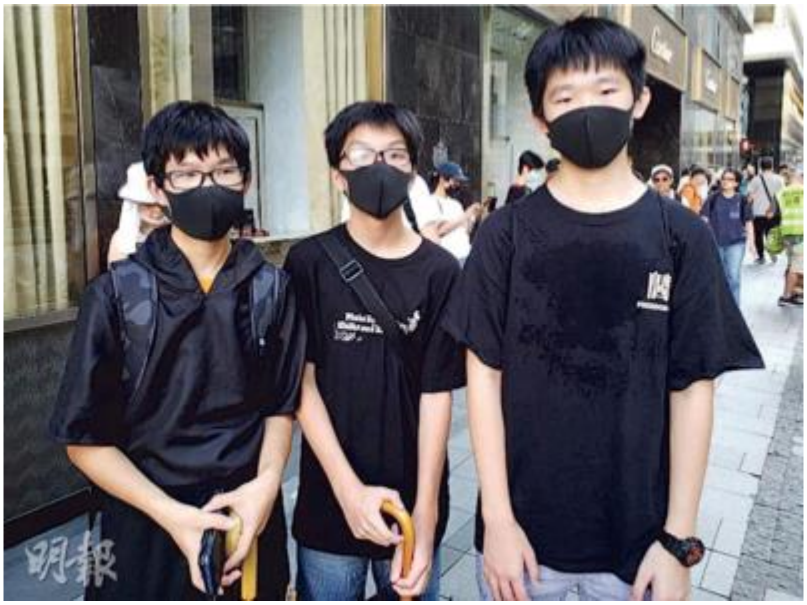


Figure 1 - Sample record 1

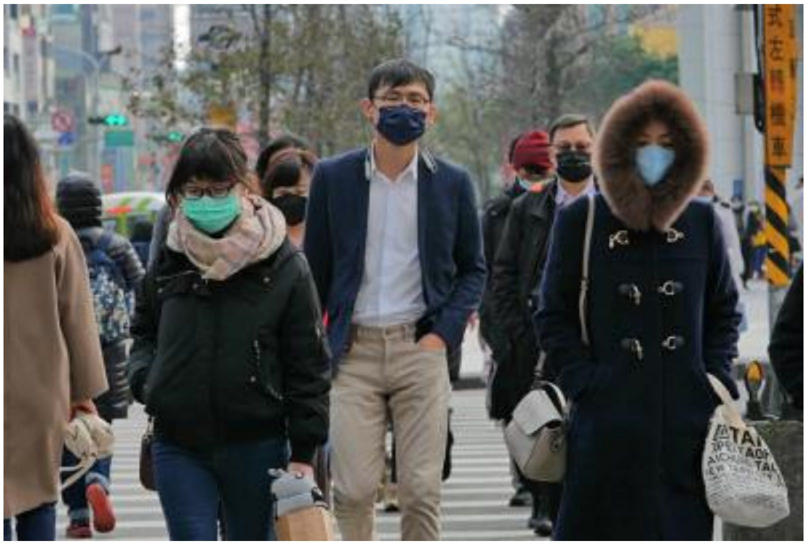


Figure 2 - Sample record 2

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:

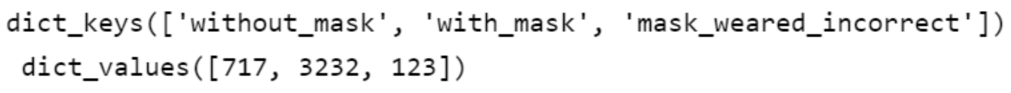
```
img_names=[]
xml_names=[]
for dirname, _ , filenames in os.walk('Data'):
    for filename in filenames:
        if os.path.join(dirname, filename)[-3:]!="xml":
            img_names.append(filename)
        else:
            xml_names.append(filename)
```

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:



```
dict_keys(['without_mask', 'with_mask', 'mask_weared_incorrect'])
 dict_values([717, 3232, 123])
```

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()
```

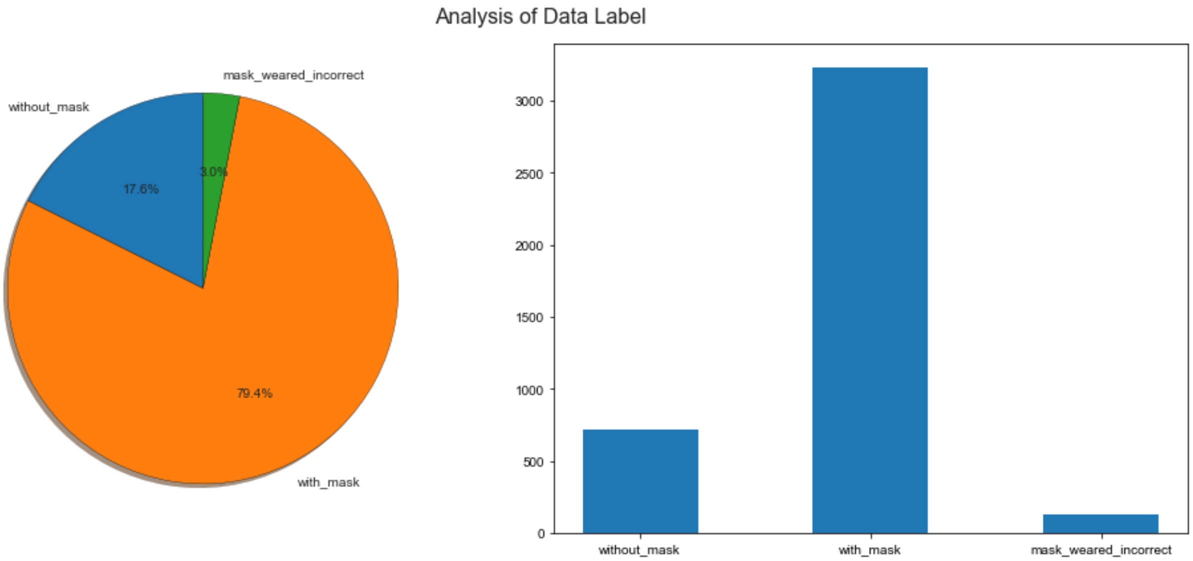


Figure 4 - Data visualization of the frequency of classes

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:

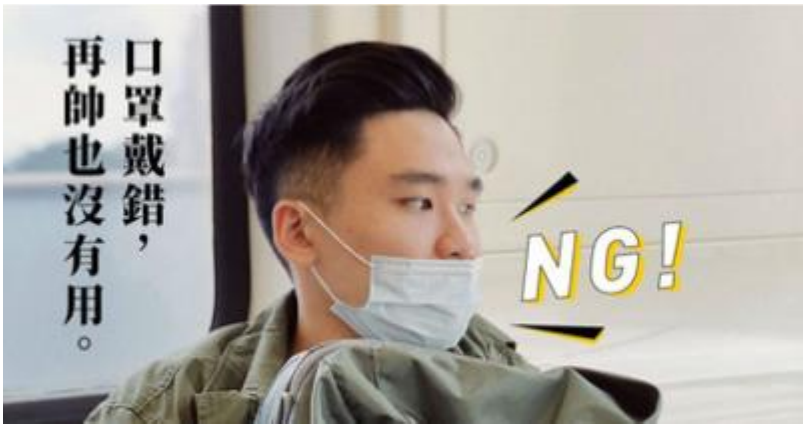


Figure 5 - Raw images without faces identified

And we have to use XML files to identify them:

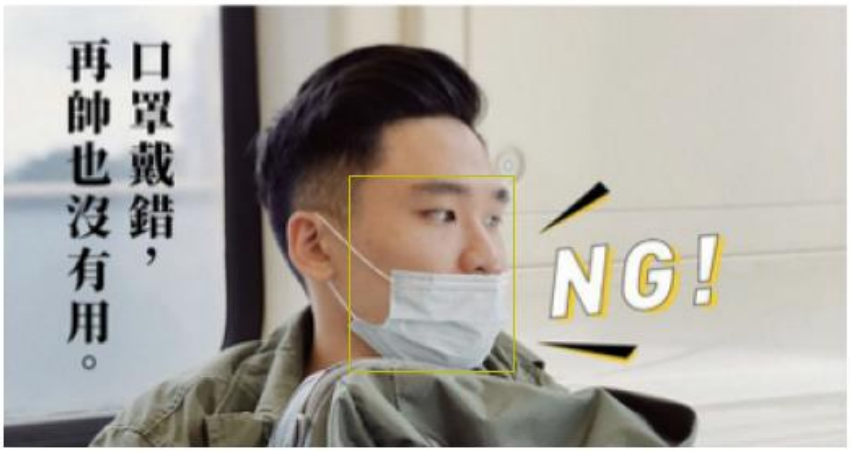


Figure 6 - Faces identified and labeled

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])
```

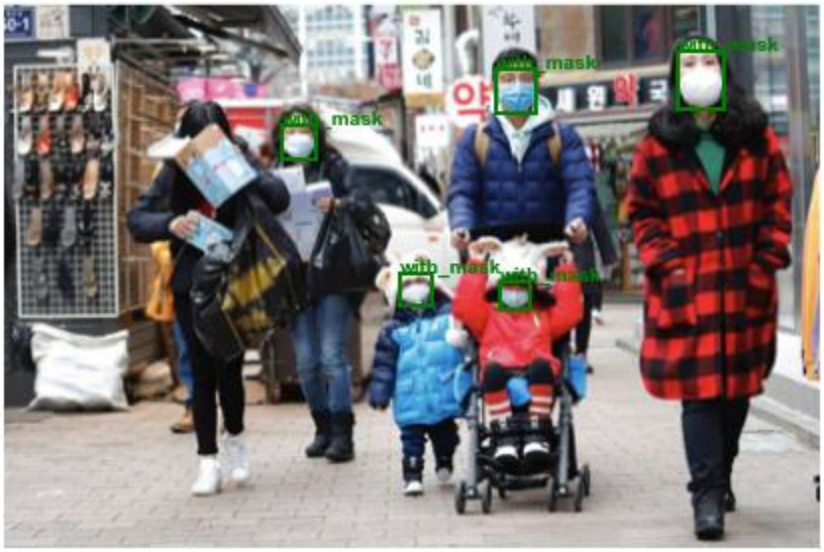


Figure 7 - Random image with faces identified No. 1

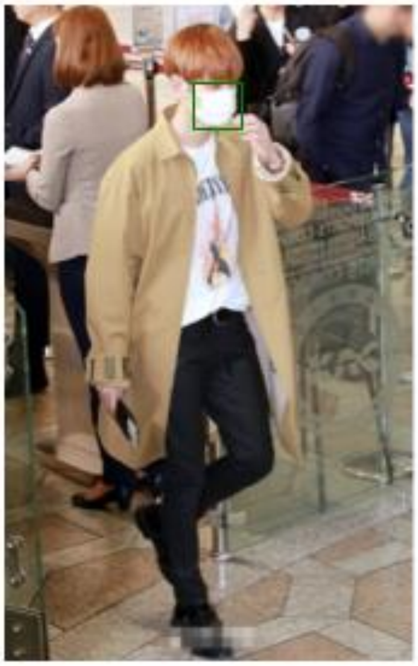


Figure 8 - Random image with faces identified No. 2

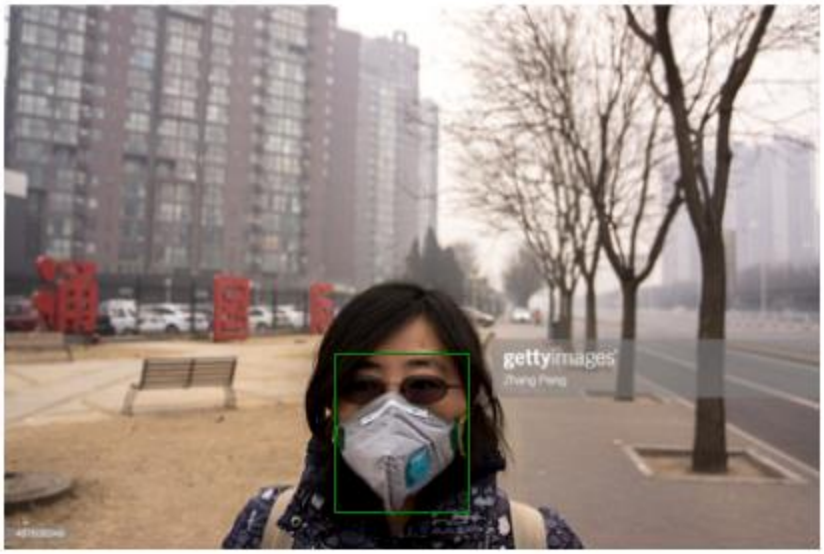


Figure 9 - Random image with faces identified No. 3

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("RG
B"), 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()
```

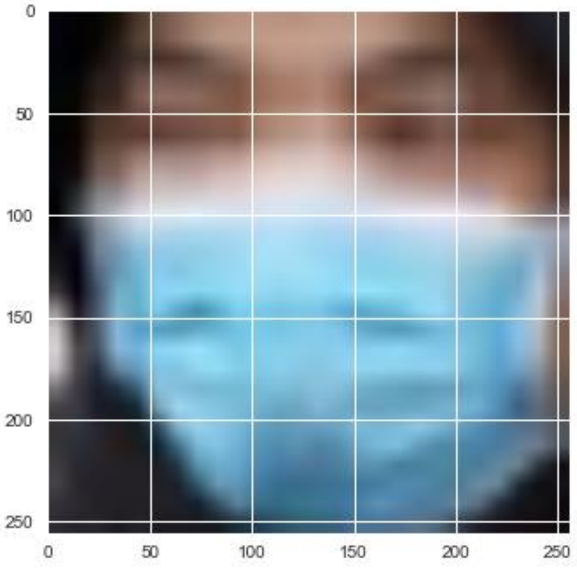


Figure 10 - Cropped images No. 1

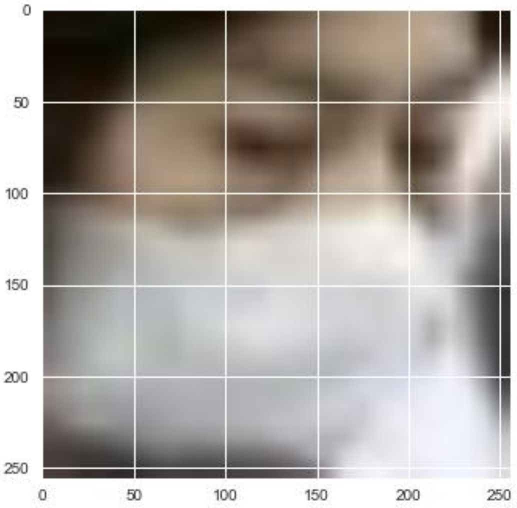


Figure 11 - Cropped images No. 2

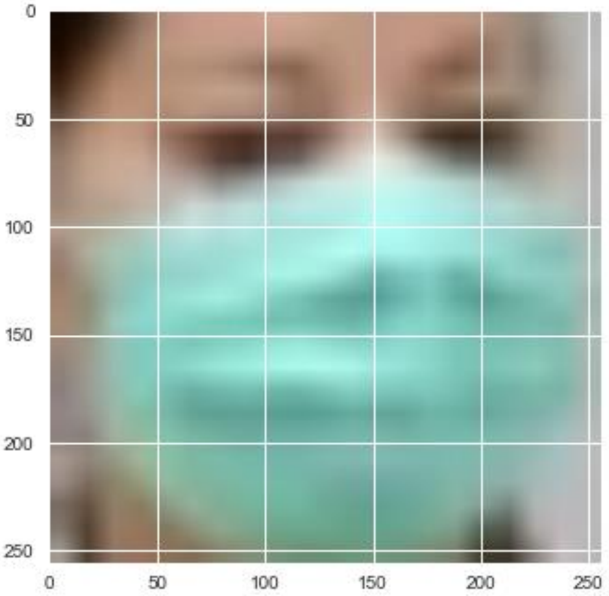


Figure 12 - Cropped images No. 3

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)
```

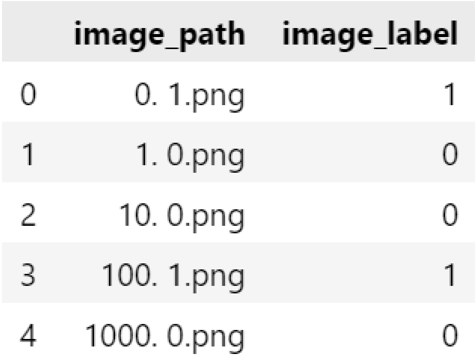
8. Splitting Dataset into Train/Dev/Test Sets

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()
```



	image_path	image_label
0	0. 1.png	1
1	1. 0.png	0
2	10. 0.png	0
3	100. 1.png	1
4	1000. 0.png	0

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()
```

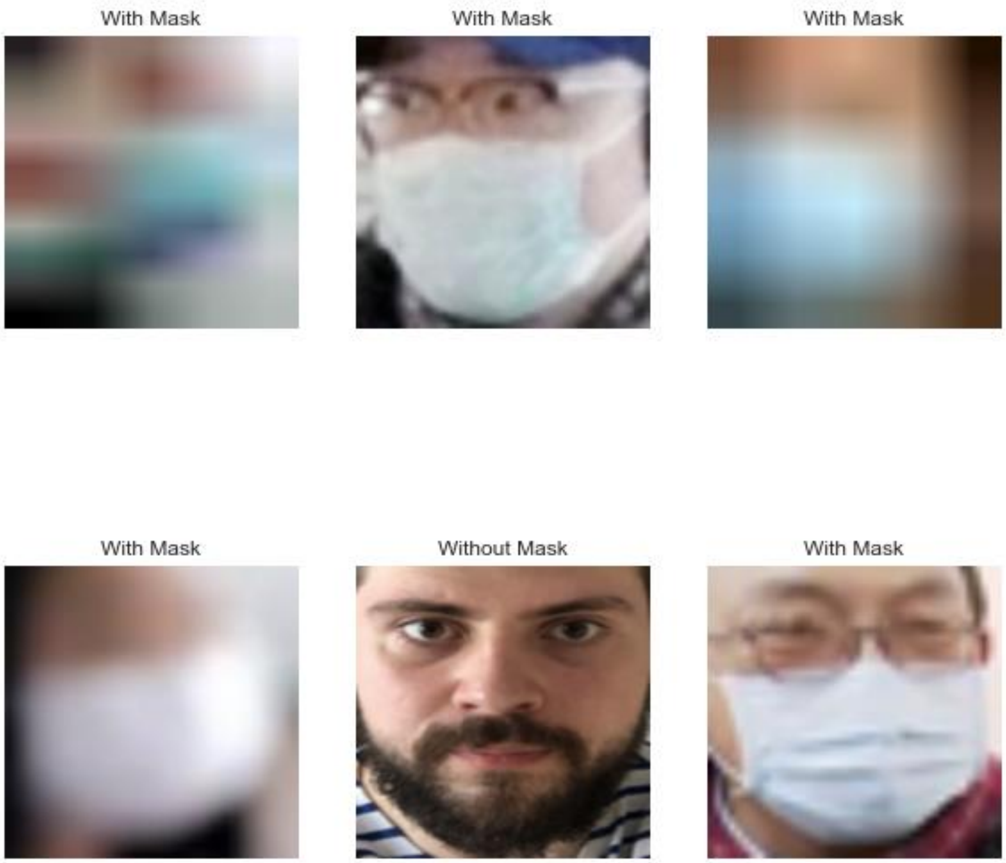


Figure 14 - Instances of the test set

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.

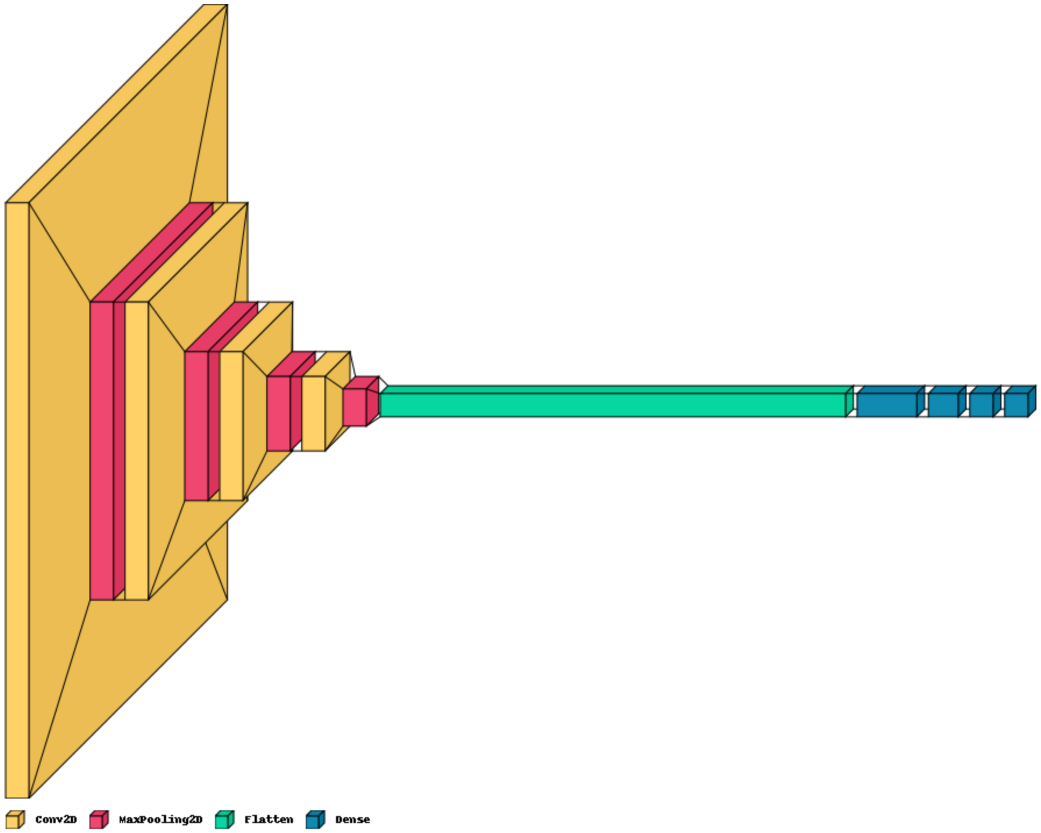


Figure 15 - First model's architecture

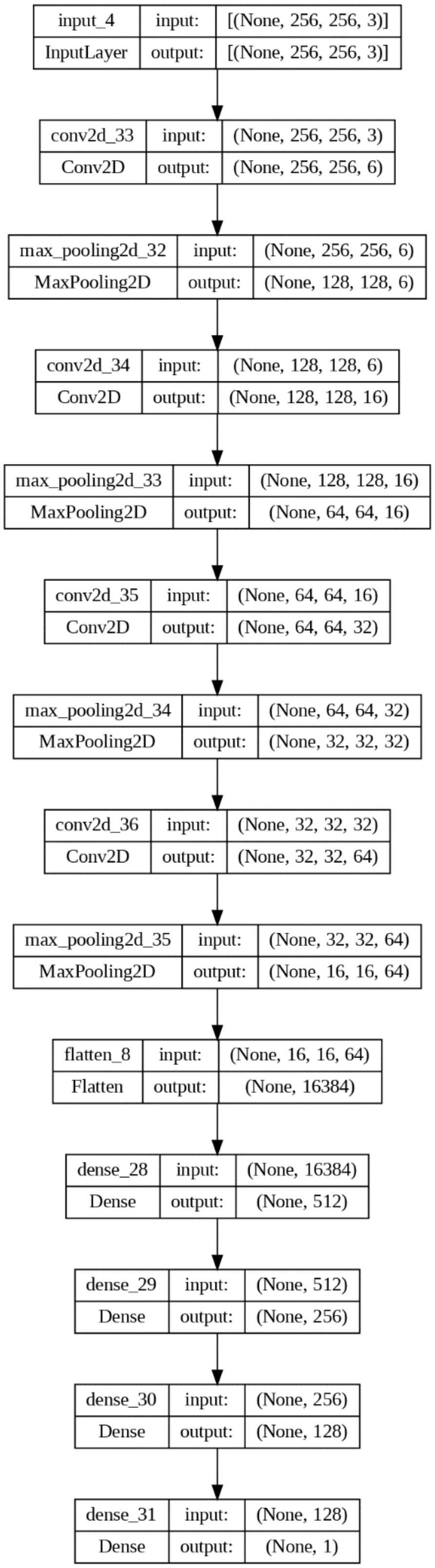


Figure 16 - First model's dot format architecture

```
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.

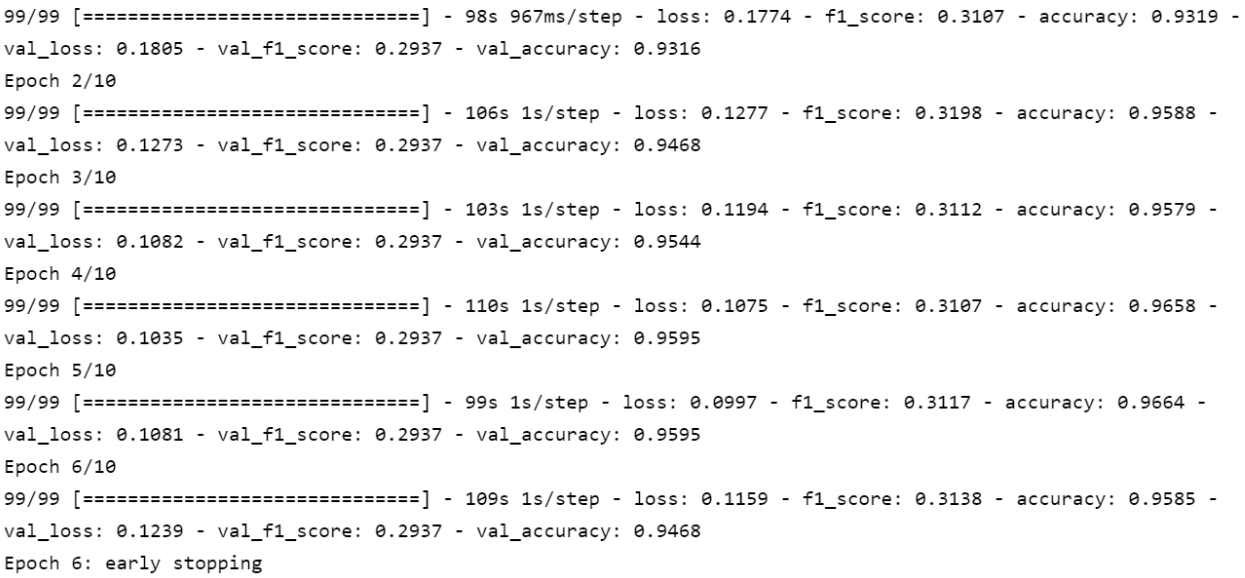
```
save_dir = "./results/"
model_name = '4CNN-3FC_Simple.h5'
model_path = os.path.join(save_dir, model_name)

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)
          ])
```

☑ 4CNN-3FC_Simple.h5 1/15/2023 12:00 AM H5 File 101,094 KB

Figure 17 - First model's output

And the training stage is shown below:



```
99/99 [==============================] - 98s 967ms/step - loss: 0.1774 - f1_score: 0.3107 - accuracy: 0.9319 -
val_loss: 0.1805 - val_f1_score: 0.2937 - val_accuracy: 0.9316
Epoch 2/10
99/99 [==============================] - 106s 1s/step - loss: 0.1277 - f1_score: 0.3198 - accuracy: 0.9588 -
val_loss: 0.1273 - val_f1_score: 0.2937 - val_accuracy: 0.9468
Epoch 3/10
99/99 [==============================] - 103s 1s/step - loss: 0.1194 - f1_score: 0.3112 - accuracy: 0.9579 -
val_loss: 0.1082 - val_f1_score: 0.2937 - val_accuracy: 0.9544
Epoch 4/10
99/99 [==============================] - 110s 1s/step - loss: 0.1075 - f1_score: 0.3107 - accuracy: 0.9658 -
val_loss: 0.1035 - val_f1_score: 0.2937 - val_accuracy: 0.9595
Epoch 5/10
99/99 [==============================] - 99s 1s/step - loss: 0.0997 - f1_score: 0.3117 - accuracy: 0.9664 -
val_loss: 0.1081 - val_f1_score: 0.2937 - val_accuracy: 0.9595
Epoch 6/10
99/99 [==============================] - 109s 1s/step - loss: 0.1159 - f1_score: 0.3138 - accuracy: 0.9585 -
val_loss: 0.1239 - val_f1_score: 0.2937 - val_accuracy: 0.9468
Epoch 6: early stopping
```

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()
```

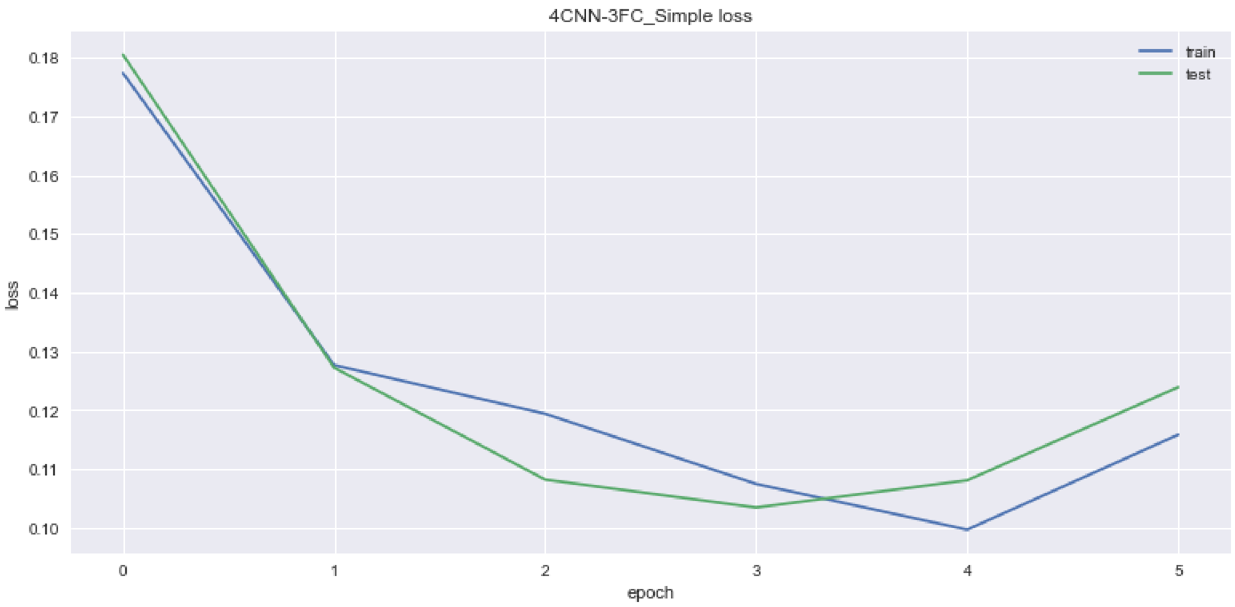


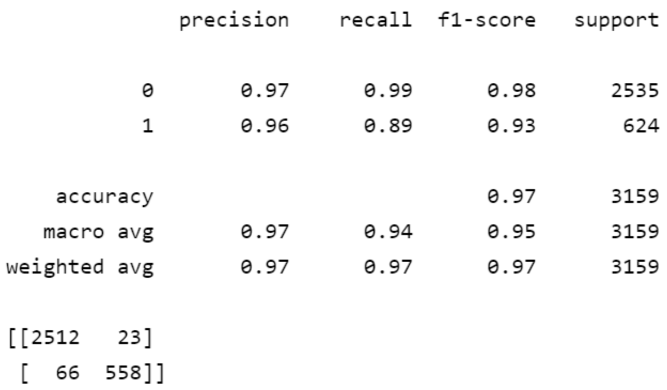
Figure 19 - Train and dev set loss for each epoch

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.



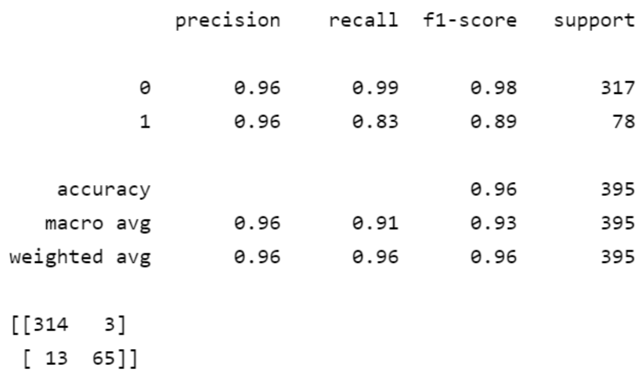
```
              precision    recall  f1-score   support

           0       0.97      0.99      0.98      2535
           1       0.96      0.89      0.93       624

    accuracy                           0.97      3159
   macro avg       0.97      0.94      0.95      3159
weighted avg       0.97      0.97      0.97      3159

[[2512   23]
 [  66  558]]
```

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



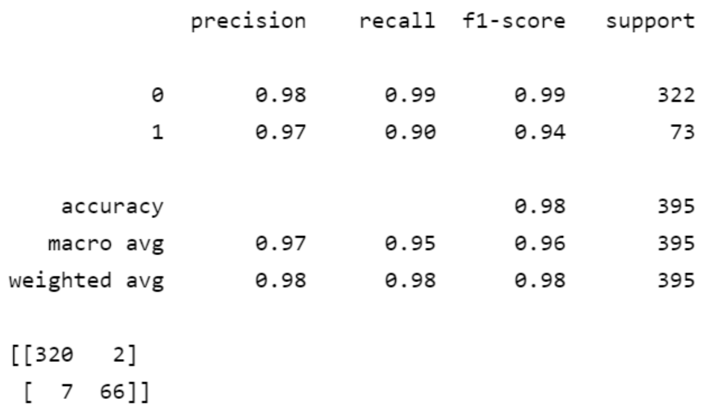
```
              precision    recall  f1-score   support

           0       0.96      0.99      0.98       317
           1       0.96      0.83      0.89        78

    accuracy                           0.96       395
   macro avg       0.96      0.91      0.93       395
weighted avg       0.96      0.96      0.96       395

[[314   3]
 [ 13  65]]
```

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



```
              precision    recall  f1-score   support

           0       0.98      0.99      0.99       322
           1       0.97      0.90      0.94        73

    accuracy                           0.98       395
   macro avg       0.97      0.95      0.96       395
weighted avg       0.98      0.98      0.98       395

[[320   2]
 [  7  66]]
```

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:

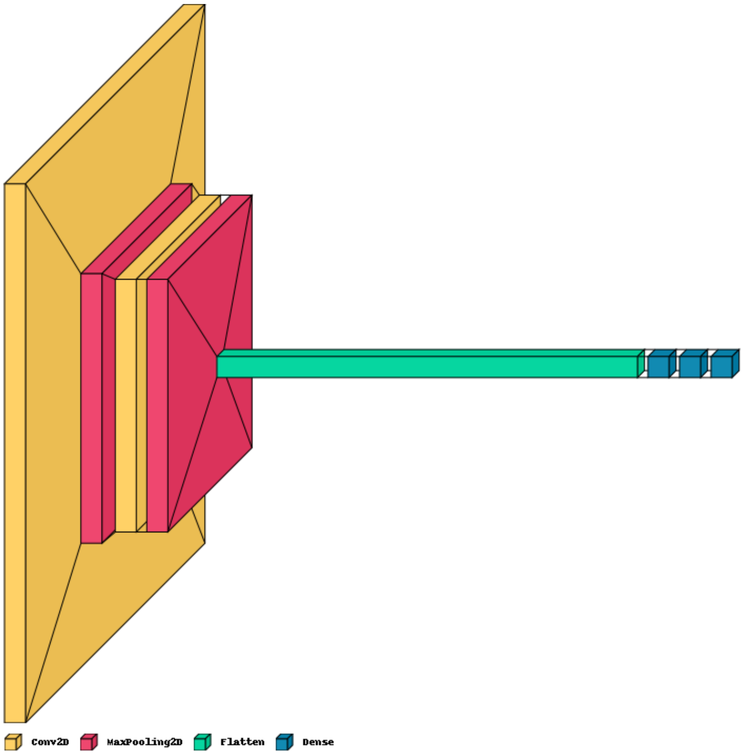


Figure 23 – Second model's architecture

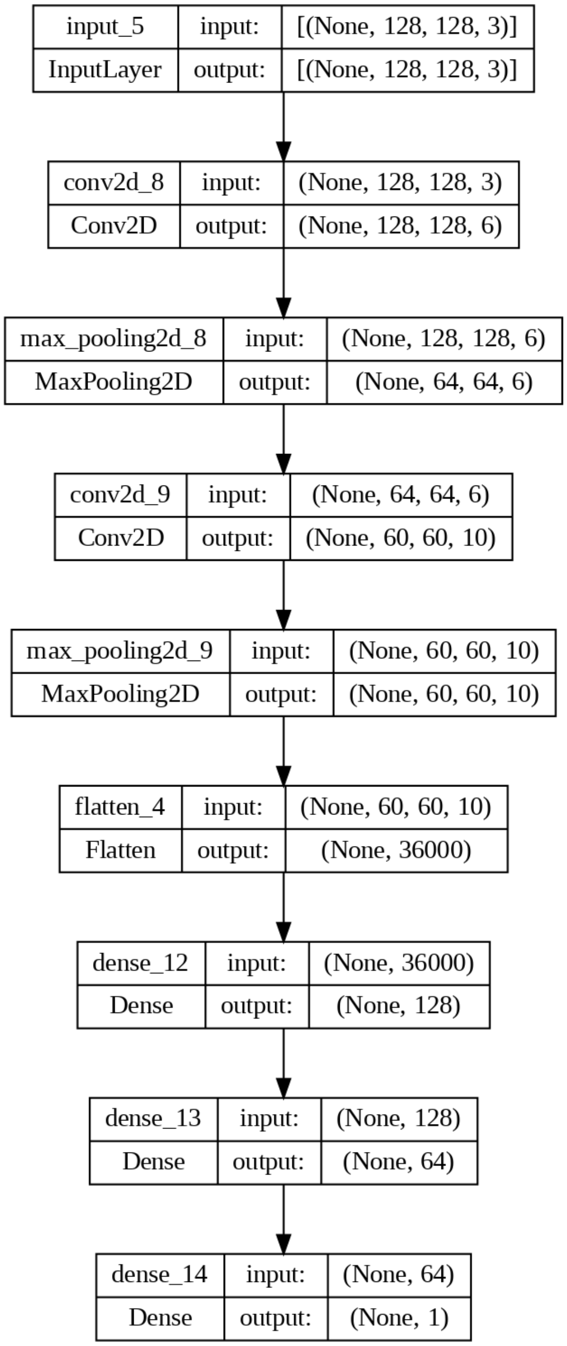


Figure 24 - Second model's dot format architecture

```
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.

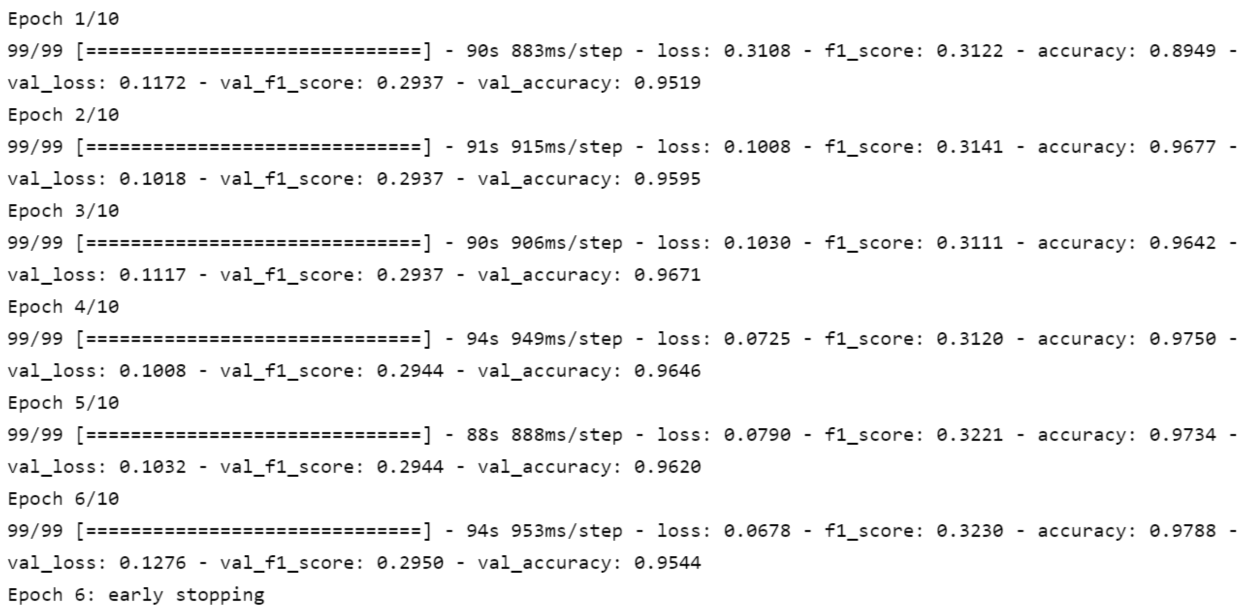
```
save_dir = "./results/"
model_name = '2CNN-2FC_Simple.h5'
model_path = os.path.join(save_dir, model_name)

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)
          ])
```

☑ 2CNN-2FC_Simple.h5   1/15/2023 12:19 AM   H5 File   230,812 KB

Figure 25 - Second model's output

And the training stage is shown below:



```
Epoch 1/10
99/99 [==============================] - 90s 883ms/step - loss: 0.3108 - f1_score: 0.3122 - accuracy: 0.8949 -
val_loss: 0.1172 - val_f1_score: 0.2937 - val_accuracy: 0.9519
Epoch 2/10
99/99 [==============================] - 91s 915ms/step - loss: 0.1008 - f1_score: 0.3141 - accuracy: 0.9677 -
val_loss: 0.1018 - val_f1_score: 0.2937 - val_accuracy: 0.9595
Epoch 3/10
99/99 [==============================] - 90s 906ms/step - loss: 0.1030 - f1_score: 0.3111 - accuracy: 0.9642 -
val_loss: 0.1117 - val_f1_score: 0.2937 - val_accuracy: 0.9671
Epoch 4/10
99/99 [==============================] - 94s 949ms/step - loss: 0.0725 - f1_score: 0.3120 - accuracy: 0.9750 -
val_loss: 0.1008 - val_f1_score: 0.2944 - val_accuracy: 0.9646
Epoch 5/10
99/99 [==============================] - 88s 888ms/step - loss: 0.0790 - f1_score: 0.3221 - accuracy: 0.9734 -
val_loss: 0.1032 - val_f1_score: 0.2944 - val_accuracy: 0.9620
Epoch 6/10
99/99 [==============================] - 94s 953ms/step - loss: 0.0678 - f1_score: 0.3230 - accuracy: 0.9788 -
val_loss: 0.1276 - val_f1_score: 0.2950 - val_accuracy: 0.9544
Epoch 6: early stopping
```

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:

```python
# 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()
```

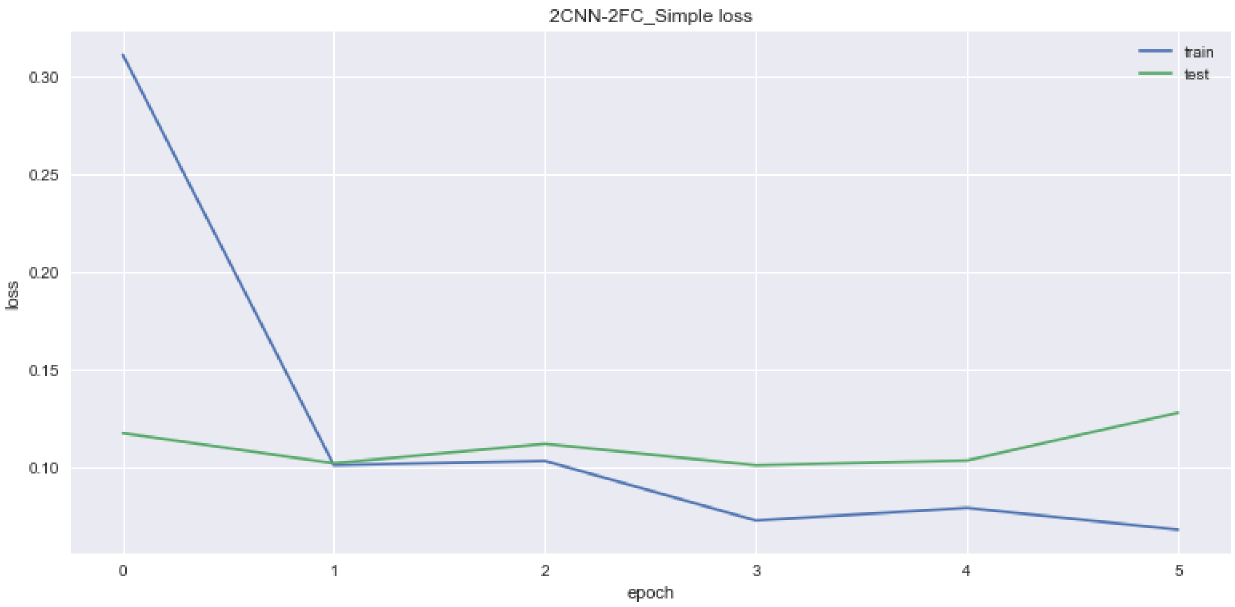


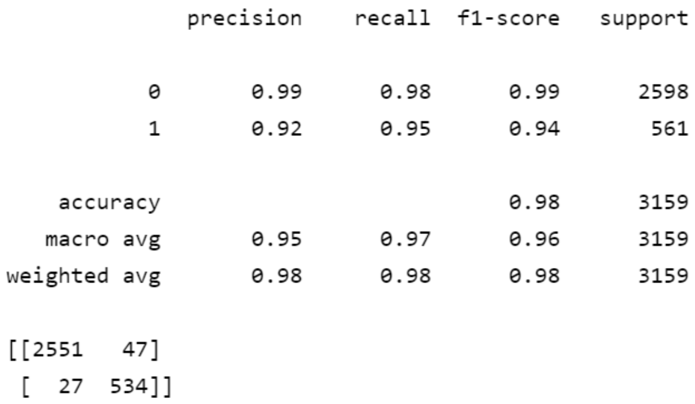
Figure 27 - Train and dev set loss for each epoch

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



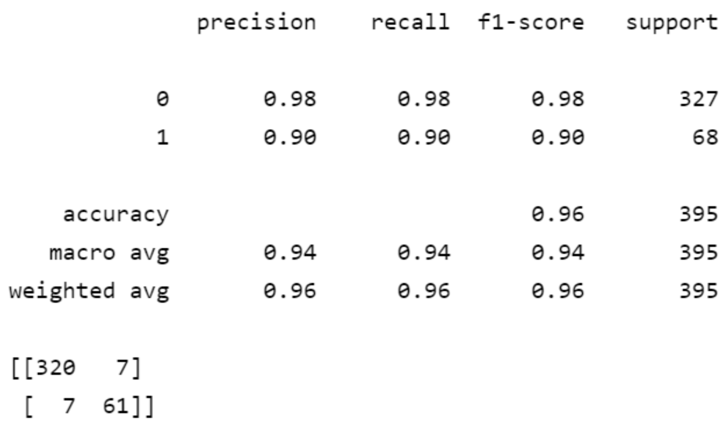
```
              precision    recall  f1-score   support

           0       0.99      0.98      0.99      2598
           1       0.92      0.95      0.94       561

    accuracy                           0.98      3159
   macro avg       0.95      0.97      0.96      3159
weighted avg       0.98      0.98      0.98      3159

[[2551   47]
 [  27  534]]
```

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



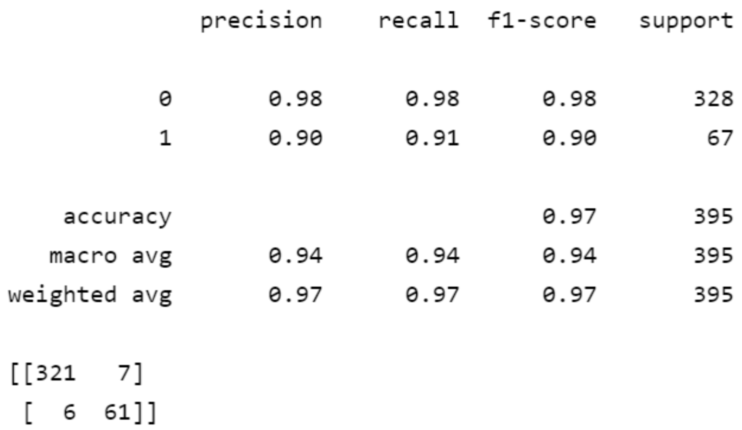
```
              precision    recall  f1-score   support

           0       0.98      0.98      0.98       327
           1       0.90      0.90      0.90        68

    accuracy                           0.96       395
   macro avg       0.94      0.94      0.94       395
weighted avg       0.96      0.96      0.96       395

[[320   7]
 [  7  61]]
```

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



```
              precision    recall  f1-score   support

           0       0.98      0.98      0.98       328
           1       0.90      0.91      0.90        67

    accuracy                           0.97       395
   macro avg       0.94      0.94      0.94       395
weighted avg       0.97      0.97      0.97       395

[[321   7]
 [  6  61]]
```

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:

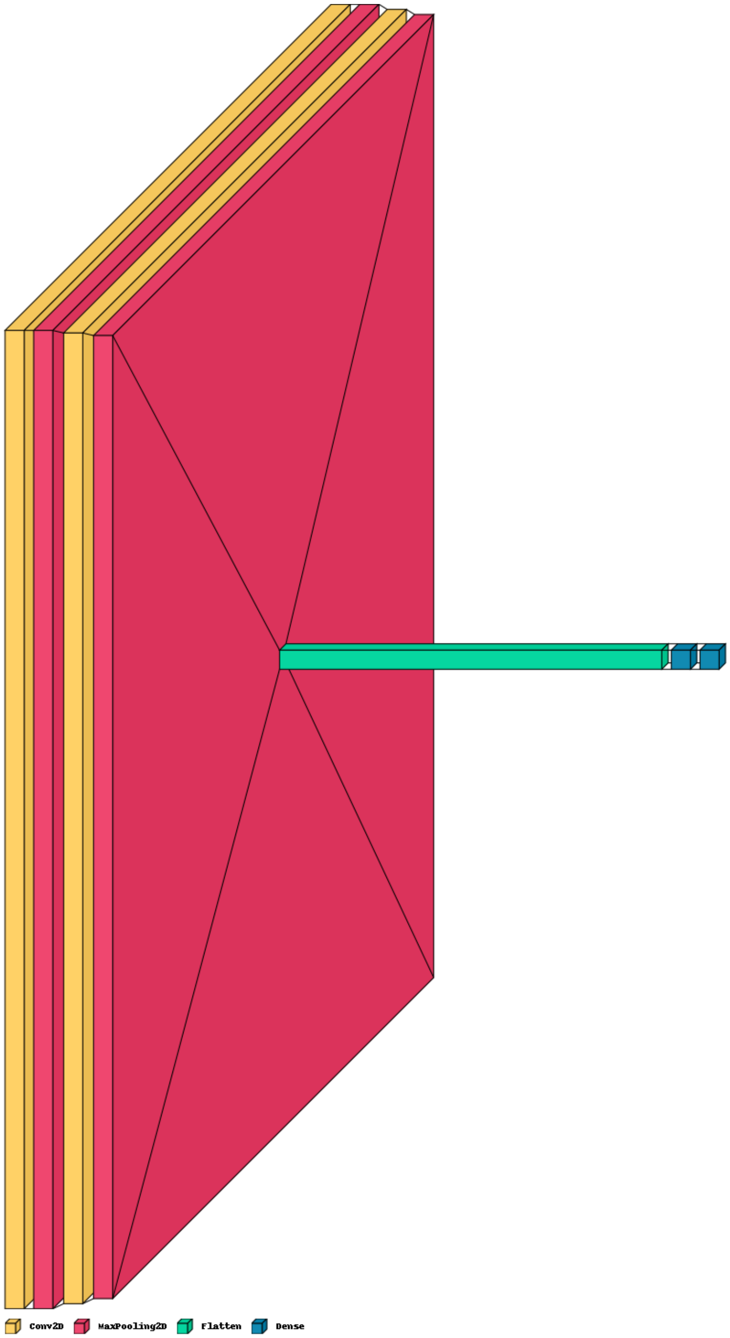


Figure 31 – Third model's architecture

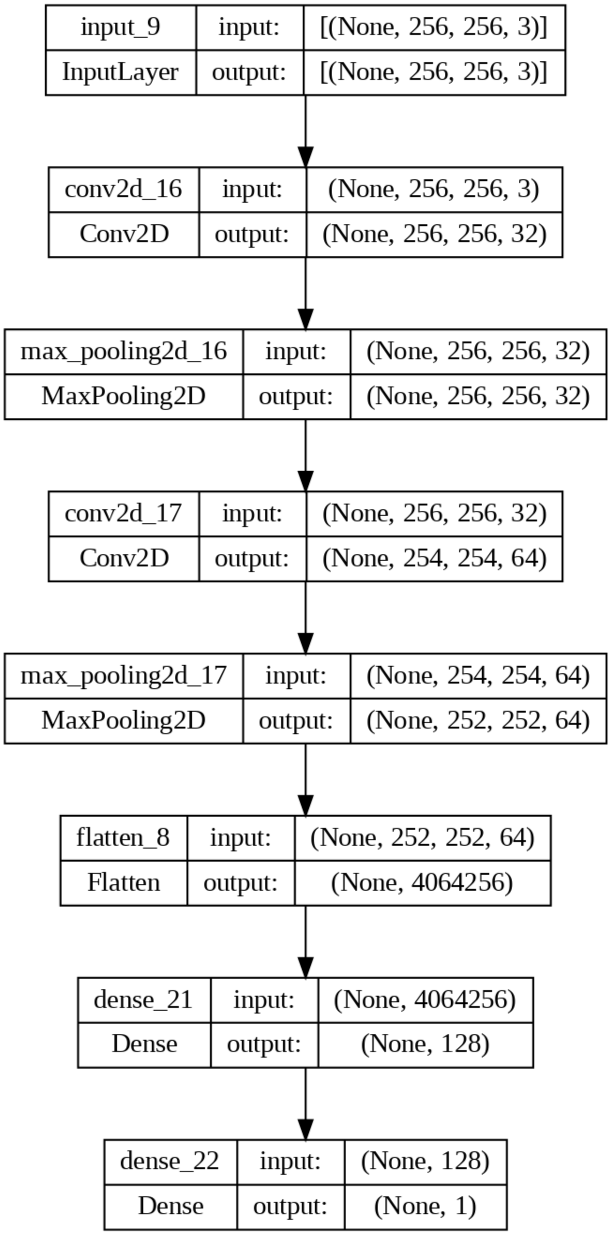


Figure 32 - Third model's dot format architecture

```
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.

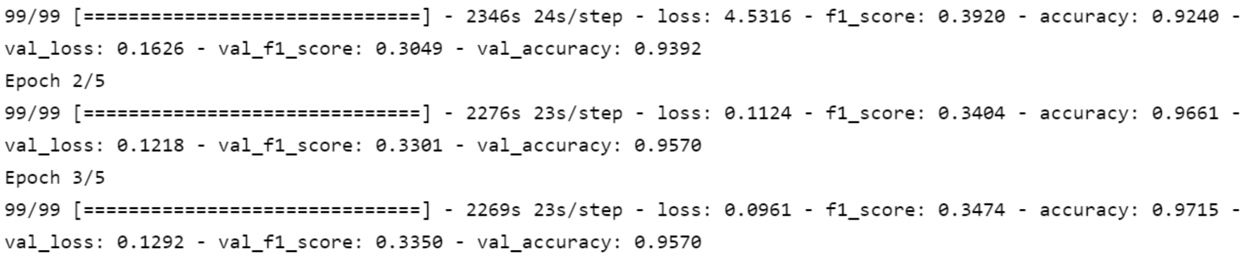
```
save_dir = "./results/"
model_name = '2CNN-1FC_Simple.h5'
model_path = os.path.join(save_dir, model_name)

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)
     ])
```

☑ 2CNN-1FC_Simple.h5 1/15/2023 2:42 AM H5 File 6,096,657 KB

Figure 33 - Second model's output

And the training stage is shown below:



```
99/99 [==============================] - 2346s 24s/step - loss: 4.5316 - f1_score: 0.3920 - accuracy: 0.9240 -
val_loss: 0.1626 - val_f1_score: 0.3049 - val_accuracy: 0.9392
Epoch 2/5
99/99 [==============================] - 2276s 23s/step - loss: 0.1124 - f1_score: 0.3404 - accuracy: 0.9661 -
val_loss: 0.1218 - val_f1_score: 0.3301 - val_accuracy: 0.9570
Epoch 3/5
99/99 [==============================] - 2269s 23s/step - loss: 0.0961 - f1_score: 0.3474 - accuracy: 0.9715 -
val_loss: 0.1292 - val_f1_score: 0.3350 - val_accuracy: 0.9570
```

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()
```

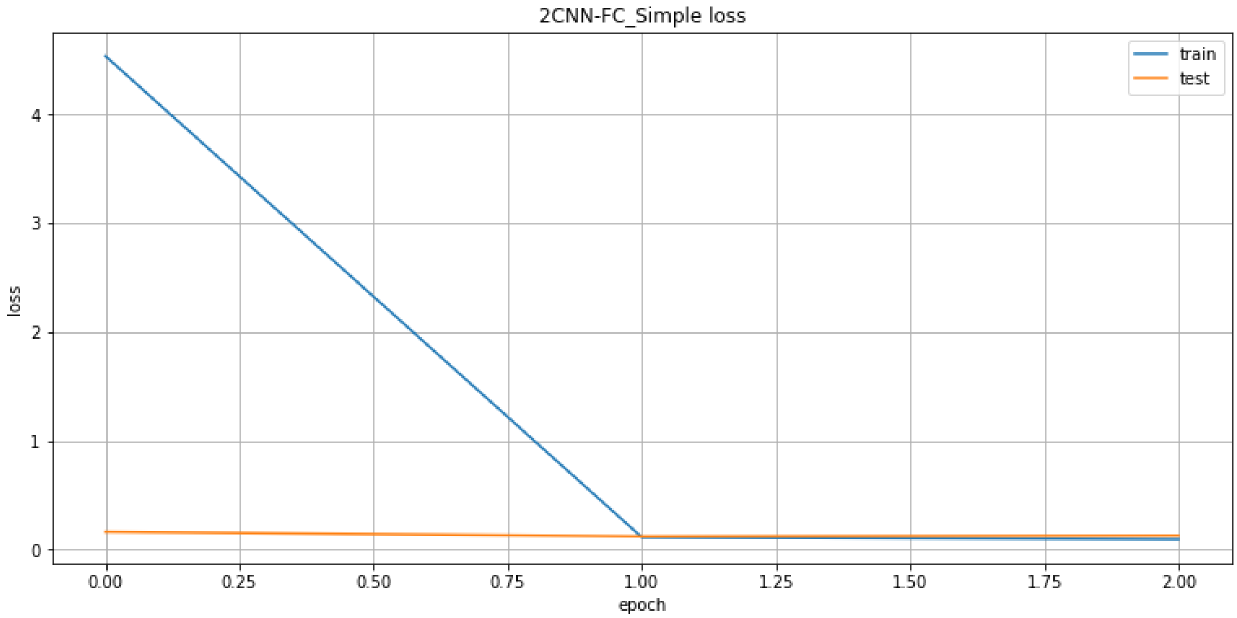


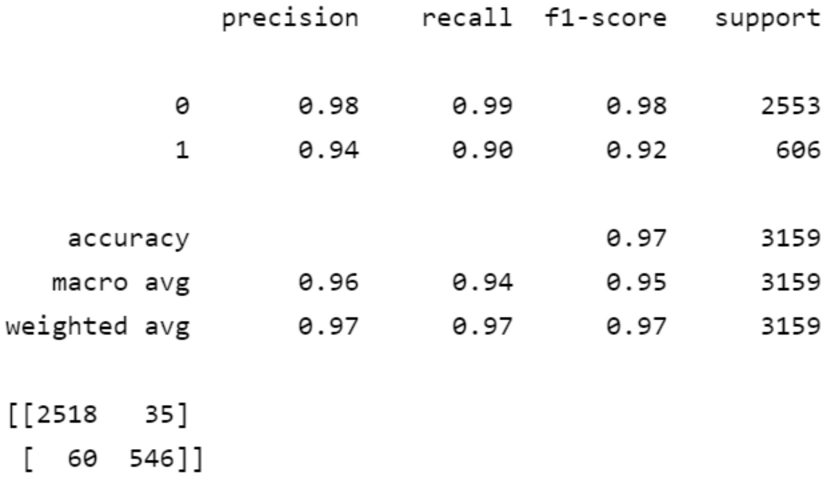
Figure 35 - Train and dev set loss for each epoch

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:



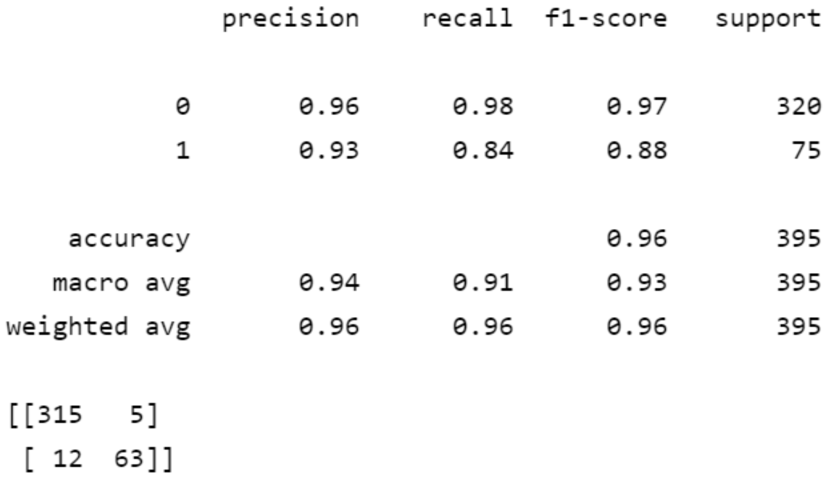
```
              precision    recall  f1-score   support

           0       0.98      0.99      0.98      2553
           1       0.94      0.90      0.92       606

    accuracy                           0.97      3159
   macro avg       0.96      0.94      0.95      3159
weighted avg       0.97      0.97      0.97      3159

[[2518   35]
 [  60  546]]
```

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



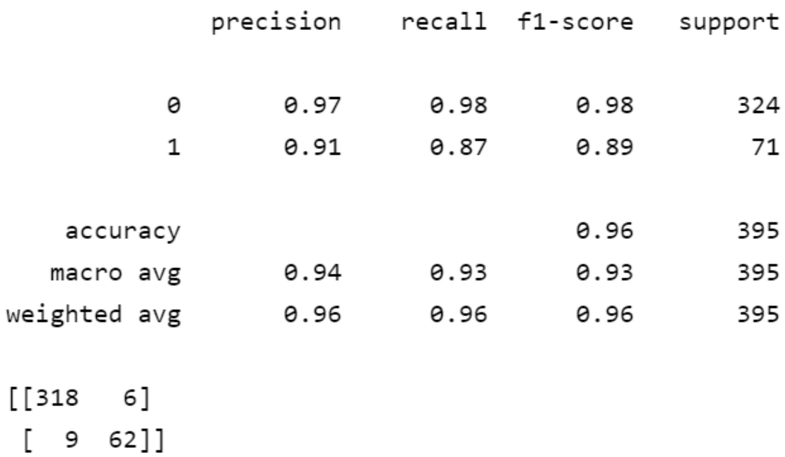
```
              precision    recall  f1-score   support

           0       0.96      0.98      0.97       320
           1       0.93      0.84      0.88        75

    accuracy                           0.96       395
   macro avg       0.94      0.91      0.93       395
weighted avg       0.96      0.96      0.96       395

[[315   5]
 [ 12  63]]
```

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



```
              precision    recall  f1-score   support

           0       0.97      0.98      0.98       324
           1       0.91      0.87      0.89        71

    accuracy                           0.96       395
   macro avg       0.94      0.93      0.93       395
weighted avg       0.96      0.96      0.96       395

[[318   6]
 [  9  62]]
```

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 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.prepro
cess_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.

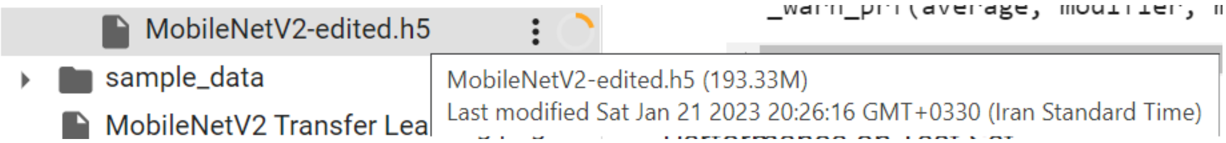
```
model.compile(optimizer='adam',
                loss=
                'binary_crossentropy',
                metrics=[tfa.metrics.F1Score(num_classes=1), 'accuracy'])
```

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)
          ])
```

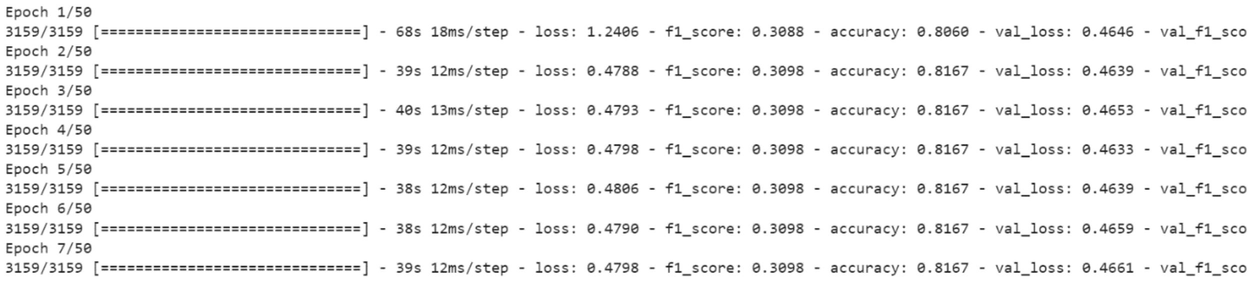


MobileNetV2-edited.h5
sample_data
MobileNetV2 Transfer Lea
MobileNetV2-edited.h5 (193.33M)
Last modified Sat Jan 21 2023 20:26:16 GMT+0330 (Iran Standard Time)

```
_warn_prf(average, modifier, m
```

Figure 40 – MobileNetV2 output

And the training stage is shown below:



```
Epoch 1/50
3159/3159 [==============================] - 68s 18ms/step - loss: 1.2406 - f1_score: 0.3088 - accuracy: 0.8060 - val_loss: 0.4646 - val_f1_sco
Epoch 2/50
3159/3159 [==============================] - 39s 12ms/step - loss: 0.4788 - f1_score: 0.3098 - accuracy: 0.8167 - val_loss: 0.4639 - val_f1_sco
Epoch 3/50
3159/3159 [==============================] - 40s 13ms/step - loss: 0.4793 - f1_score: 0.3098 - accuracy: 0.8167 - val_loss: 0.4653 - val_f1_sco
Epoch 4/50
3159/3159 [==============================] - 39s 12ms/step - loss: 0.4798 - f1_score: 0.3098 - accuracy: 0.8167 - val_loss: 0.4633 - val_f1_sco
Epoch 5/50
3159/3159 [==============================] - 38s 12ms/step - loss: 0.4806 - f1_score: 0.3098 - accuracy: 0.8167 - val_loss: 0.4639 - val_f1_sco
Epoch 6/50
3159/3159 [==============================] - 38s 12ms/step - loss: 0.4790 - f1_score: 0.3098 - accuracy: 0.8167 - val_loss: 0.4659 - val_f1_sco
Epoch 7/50
3159/3159 [==============================] - 39s 12ms/step - loss: 0.4798 - f1_score: 0.3098 - accuracy: 0.8167 - val_loss: 0.4661 - val_f1_sco
```

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)
```

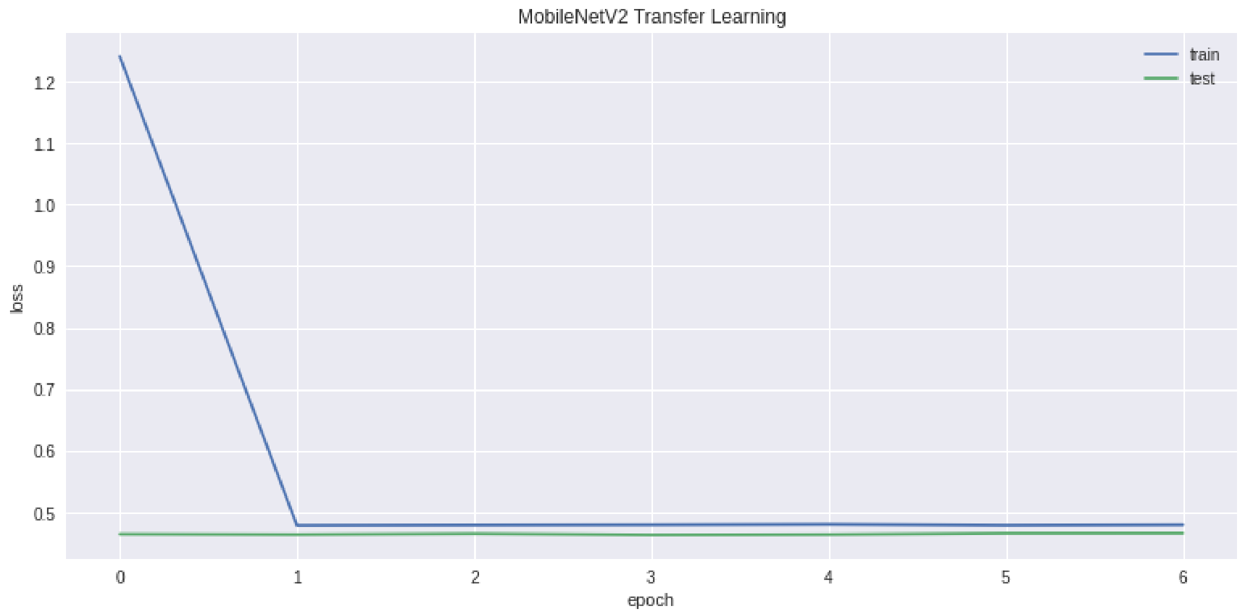


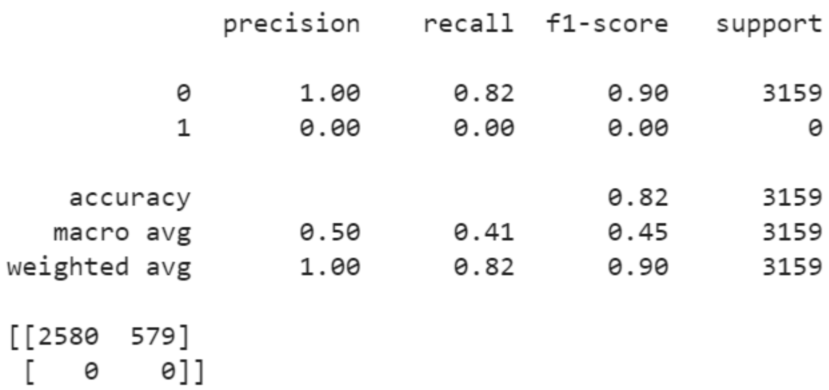
Figure 42 - Train and test set loss for each epoch

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



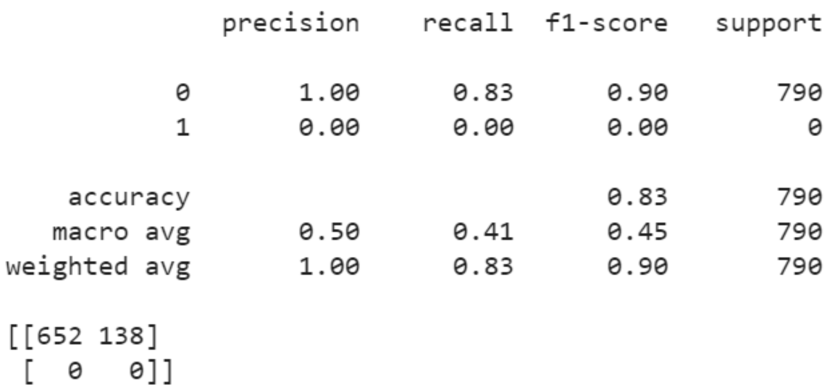
```
              precision    recall  f1-score   support

           0       1.00      0.82      0.90      3159
           1       0.00      0.00      0.00         0

    accuracy                           0.82      3159
   macro avg       0.50      0.41      0.45      3159
weighted avg       1.00      0.82      0.90      3159

[[2580  579]
 [   0    0]]
```

Figure 43 - Measurements for MobileNetV2 on train set



```
              precision    recall  f1-score   support

           0       1.00      0.83      0.90       790
           1       0.00      0.00      0.00         0

    accuracy                           0.83       790
   macro avg       0.50      0.41      0.45       790
weighted avg       1.00      0.83      0.90       790

[[652 138]
 [  0   0]]
```

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.prepr
ocess_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=headMode
l)

# 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:

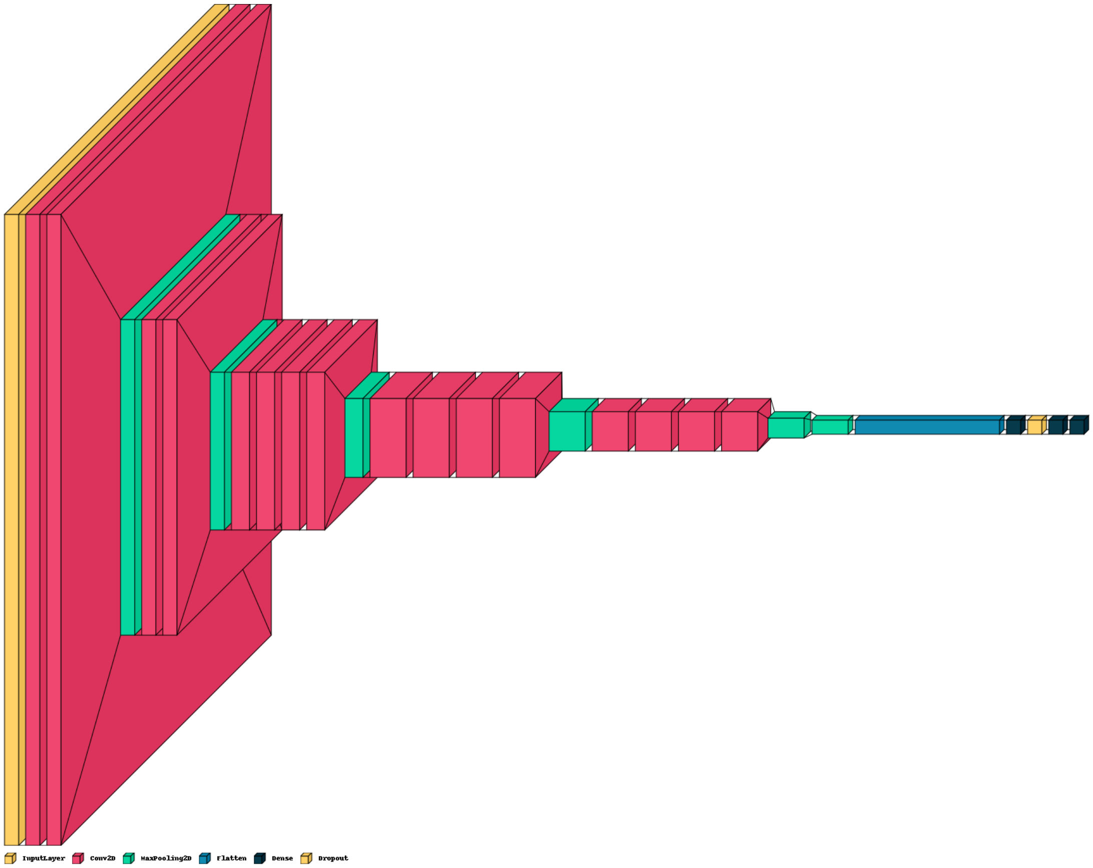


Figure 45 – VGG-19 architecture

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

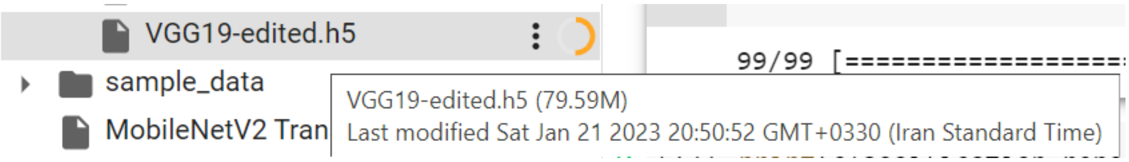
```
model_vgg.compile(optimizer='adam',
              loss=
              'binary_crossentropy',
              metrics=[tfa.metrics.F1Score(num_classes=1), 'accuracy'])
```

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)
          ])
```

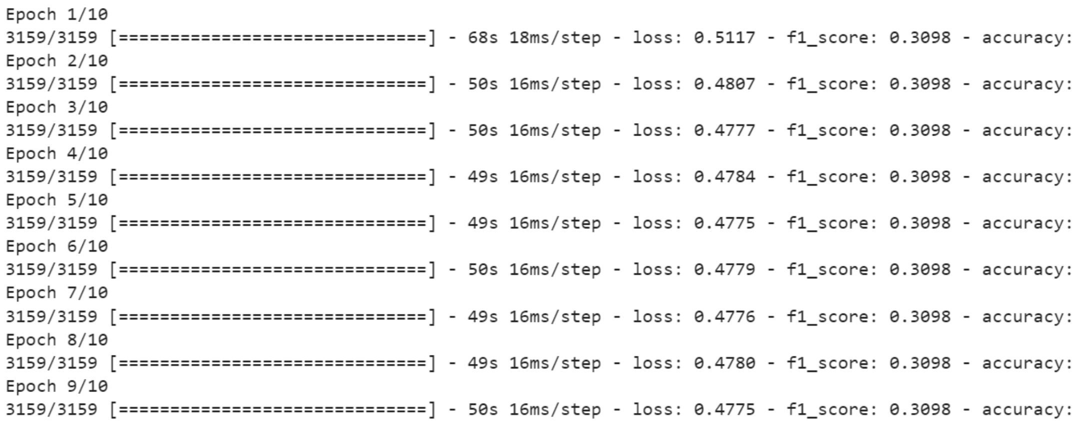


```
99/99 [==================
```

VGG19-edited.h5 (79.59M)
Last modified Sat Jan 21 2023 20:50:52 GMT+0330 (Iran Standard Time)

Figure 46 - VGG-19 output

And the training stage is shown below:



```
Epoch 1/10
3159/3159 [==============================] - 68s 18ms/step - loss: 0.5117 - f1_score: 0.3098 - accuracy:
Epoch 2/10
3159/3159 [==============================] - 50s 16ms/step - loss: 0.4807 - f1_score: 0.3098 - accuracy:
Epoch 3/10
3159/3159 [==============================] - 50s 16ms/step - loss: 0.4777 - f1_score: 0.3098 - accuracy:
Epoch 4/10
3159/3159 [==============================] - 49s 16ms/step - loss: 0.4784 - f1_score: 0.3098 - accuracy:
Epoch 5/10
3159/3159 [==============================] - 49s 16ms/step - loss: 0.4775 - f1_score: 0.3098 - accuracy:
Epoch 6/10
3159/3159 [==============================] - 50s 16ms/step - loss: 0.4779 - f1_score: 0.3098 - accuracy:
Epoch 7/10
3159/3159 [==============================] - 49s 16ms/step - loss: 0.4776 - f1_score: 0.3098 - accuracy:
Epoch 8/10
3159/3159 [==============================] - 49s 16ms/step - loss: 0.4780 - f1_score: 0.3098 - accuracy:
Epoch 9/10
3159/3159 [==============================] - 50s 16ms/step - loss: 0.4775 - f1_score: 0.3098 - accuracy:
```

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)
```

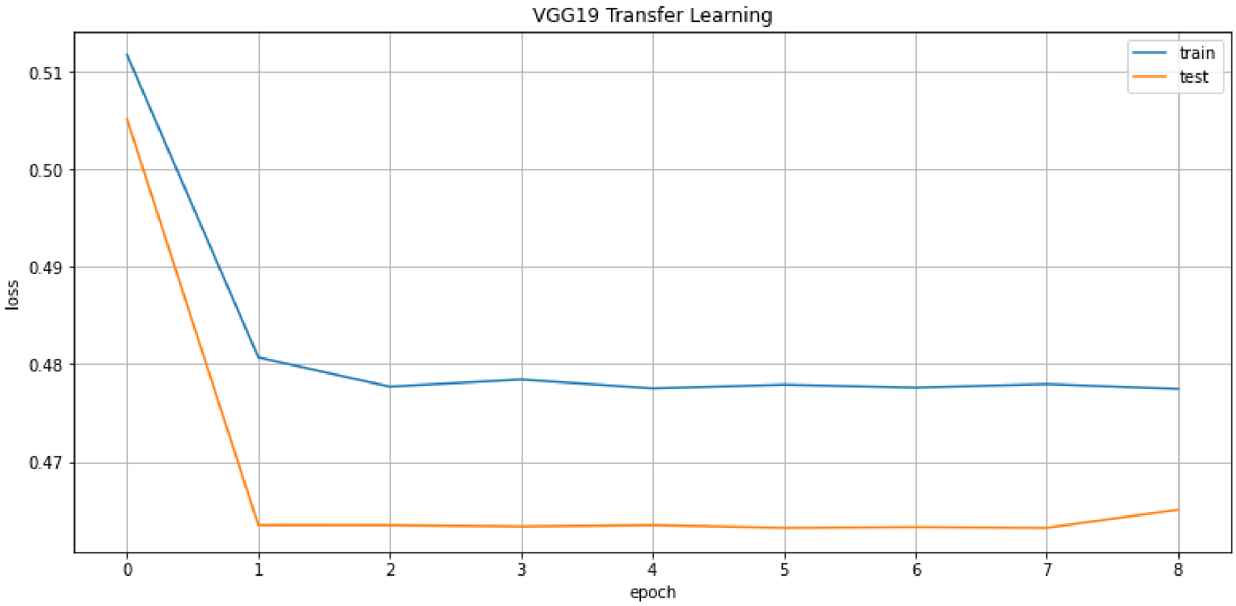


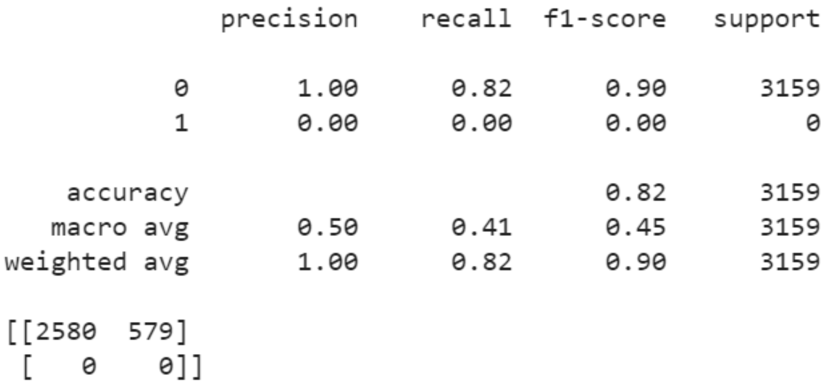
Figure 48 - Train and test set loss for each epoch

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



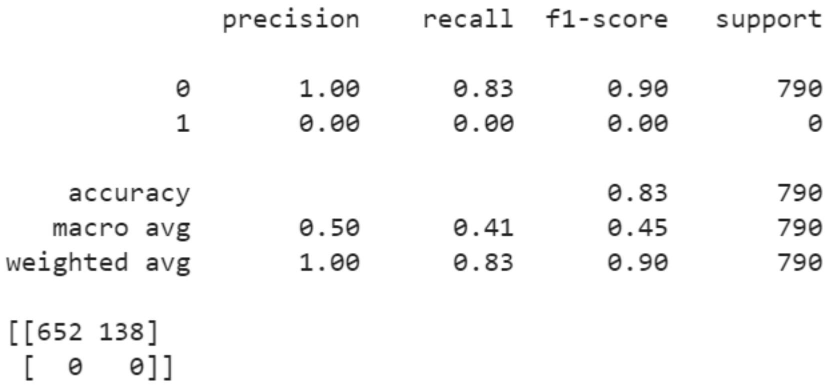
```
              precision    recall  f1-score   support

           0       1.00      0.82      0.90      3159
           1       0.00      0.00      0.00         0

    accuracy                           0.82      3159
   macro avg       0.50      0.41      0.45      3159
weighted avg       1.00      0.82      0.90      3159

[[2580  579]
 [   0    0]]
```

Figure 49 - Measurements for VGG-19 on train set



```
              precision    recall  f1-score   support

           0       1.00      0.83      0.90       790
           1       0.00      0.00      0.00         0

    accuracy                           0.83       790
   macro avg       0.50      0.41      0.45       790
weighted avg       1.00      0.83      0.90       790

[[652 138]
 [  0   0]]
```

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.pr
eprocess_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=headM
odel)

# 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.

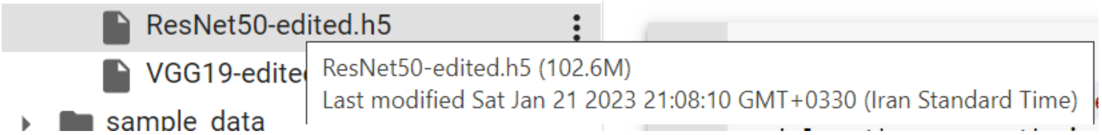
```
model_resnet.compile(optimizer='adam',
              loss=
              'binary_crossentropy',
              metrics=[tfa.metrics.F1Score(num_classes=1), 'accuracy'])
```

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)
          ])
```



ResNet50-edited.h5

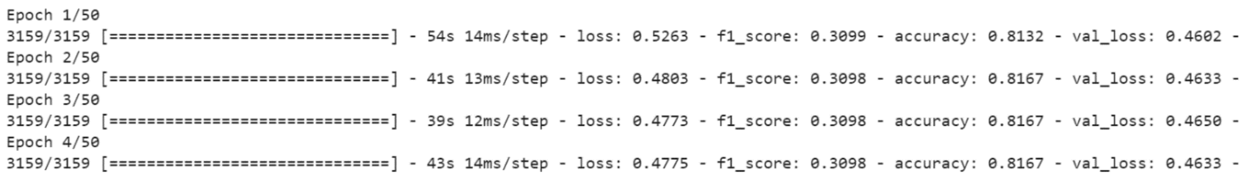
VGG19-edite

sample_data

ResNet50-edited.h5 (102.6M)
Last modified Sat Jan 21 2023 21:08:10 GMT+0330 (Iran Standard Time)

Figure 52 – ResNet-50 output

And the training stage is shown below:



```
Epoch 1/50
3159/3159 [==============================] - 54s 14ms/step - loss: 0.5263 - f1_score: 0.3099 - accuracy: 0.8132 - val_loss: 0.4602 -
Epoch 2/50
3159/3159 [==============================] - 41s 13ms/step - loss: 0.4803 - f1_score: 0.3098 - accuracy: 0.8167 - val_loss: 0.4633 -
Epoch 3/50
3159/3159 [==============================] - 39s 12ms/step - loss: 0.4773 - f1_score: 0.3098 - accuracy: 0.8167 - val_loss: 0.4650 -
Epoch 4/50
3159/3159 [==============================] - 43s 14ms/step - loss: 0.4775 - f1_score: 0.3098 - accuracy: 0.8167 - val_loss: 0.4633 -
```

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)
```

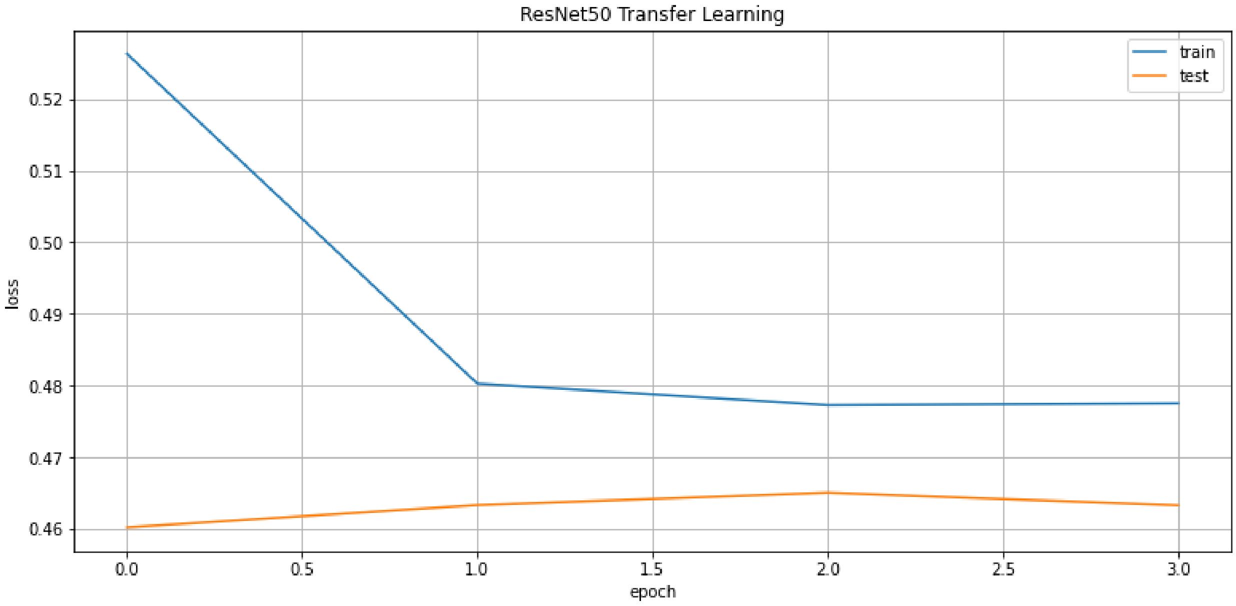


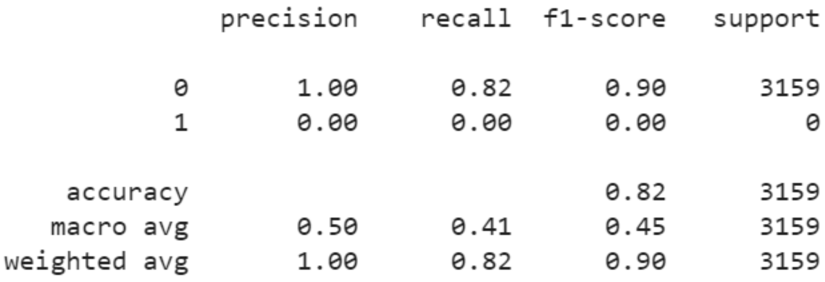
Figure 54 - Train and test set loss for each epoch

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

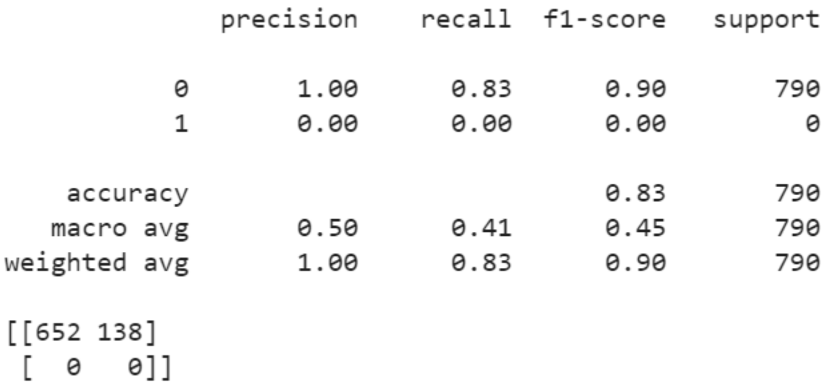


```
              precision    recall  f1-score   support

           0       1.00      0.82      0.90      3159
           1       0.00      0.00      0.00         0

    accuracy                           0.82      3159
   macro avg       0.50      0.41      0.45      3159
weighted avg       1.00      0.82      0.90      3159
```

Figure 55 - Measurements for ResNet-50 on train set



```
              precision    recall  f1-score   support

           0       1.00      0.83      0.90       790
           1       0.00      0.00      0.00         0

    accuracy                           0.83       790
   macro avg       0.50      0.41      0.45       790
weighted avg       1.00      0.83      0.90       790

[[652 138]
 [  0   0]]
```

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.incepti
on_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=he
adModel)

# 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'])
```

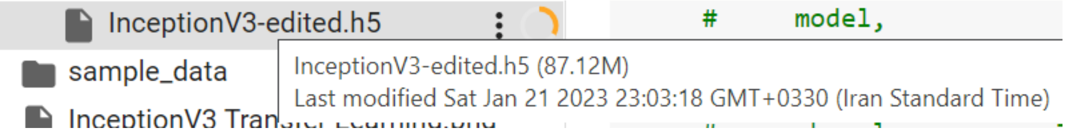
The model is then trained with a batch size equal to 1 and for 50 epochs. Although we want to run it for 50 epochs, we set an early stopping condition which may terminate the training phase sooner than expected to eschew overfitting.

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_trai
n_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)
          ])
```



InceptionV3-edited.h5

sample_data

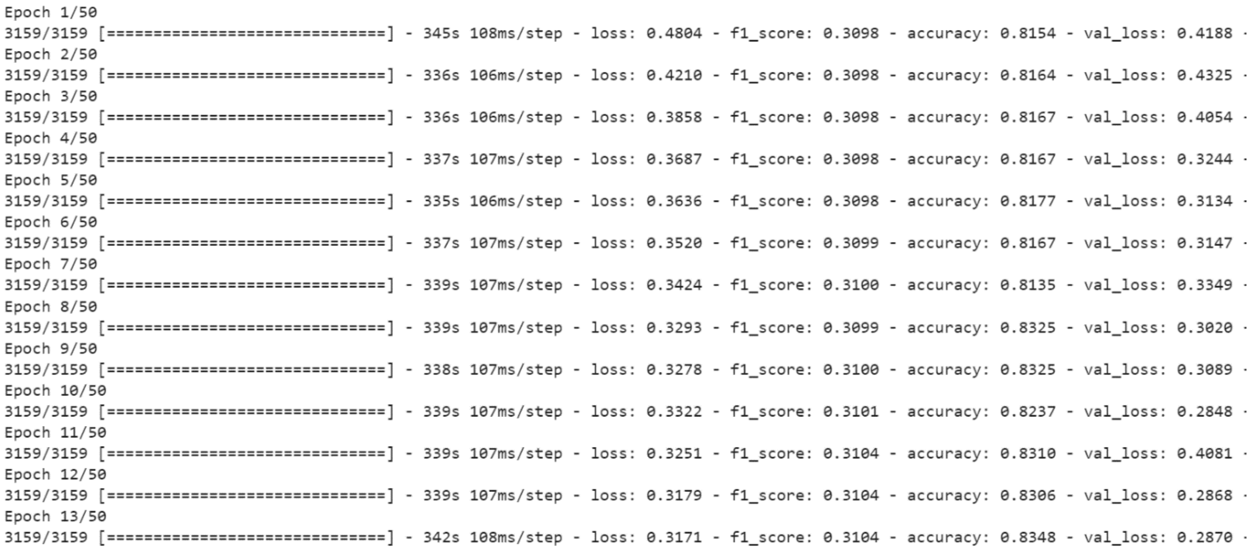
InceptionV3 Tran

```
#      model,
```

InceptionV3-edited.h5 (87.12M)
Last modified Sat Jan 21 2023 23:03:18 GMT+0330 (Iran Standard Time)

Figure 58 – Inception-V3 output

And the training stage is shown below:



```
Epoch 1/50
3159/3159 [==============================] - 345s 108ms/step - loss: 0.4804 - f1_score: 0.3098 - accuracy: 0.8154 - val_loss: 0.4188 ·
Epoch 2/50
3159/3159 [==============================] - 336s 106ms/step - loss: 0.4210 - f1_score: 0.3098 - accuracy: 0.8164 - val_loss: 0.4325 ·
Epoch 3/50
3159/3159 [==============================] - 336s 106ms/step - loss: 0.3858 - f1_score: 0.3098 - accuracy: 0.8167 - val_loss: 0.4054 ·
Epoch 4/50
3159/3159 [==============================] - 337s 107ms/step - loss: 0.3687 - f1_score: 0.3098 - accuracy: 0.8167 - val_loss: 0.3244 ·
Epoch 5/50
3159/3159 [==============================] - 335s 106ms/step - loss: 0.3636 - f1_score: 0.3098 - accuracy: 0.8177 - val_loss: 0.3134 ·
Epoch 6/50
3159/3159 [==============================] - 337s 107ms/step - loss: 0.3520 - f1_score: 0.3099 - accuracy: 0.8167 - val_loss: 0.3147 ·
Epoch 7/50
3159/3159 [==============================] - 339s 107ms/step - loss: 0.3424 - f1_score: 0.3100 - accuracy: 0.8135 - val_loss: 0.3349 ·
Epoch 8/50
3159/3159 [==============================] - 339s 107ms/step - loss: 0.3293 - f1_score: 0.3099 - accuracy: 0.8325 - val_loss: 0.3020 ·
Epoch 9/50
3159/3159 [==============================] - 338s 107ms/step - loss: 0.3278 - f1_score: 0.3100 - accuracy: 0.8325 - val_loss: 0.3089 ·
Epoch 10/50
3159/3159 [==============================] - 339s 107ms/step - loss: 0.3322 - f1_score: 0.3101 - accuracy: 0.8237 - val_loss: 0.2848 ·
Epoch 11/50
3159/3159 [==============================] - 339s 107ms/step - loss: 0.3251 - f1_score: 0.3104 - accuracy: 0.8310 - val_loss: 0.4081 ·
Epoch 12/50
3159/3159 [==============================] - 339s 107ms/step - loss: 0.3179 - f1_score: 0.3104 - accuracy: 0.8306 - val_loss: 0.2868 ·
Epoch 13/50
3159/3159 [==============================] - 342s 108ms/step - loss: 0.3171 - f1_score: 0.3104 - accuracy: 0.8348 - val_loss: 0.2870 ·
```

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)
```

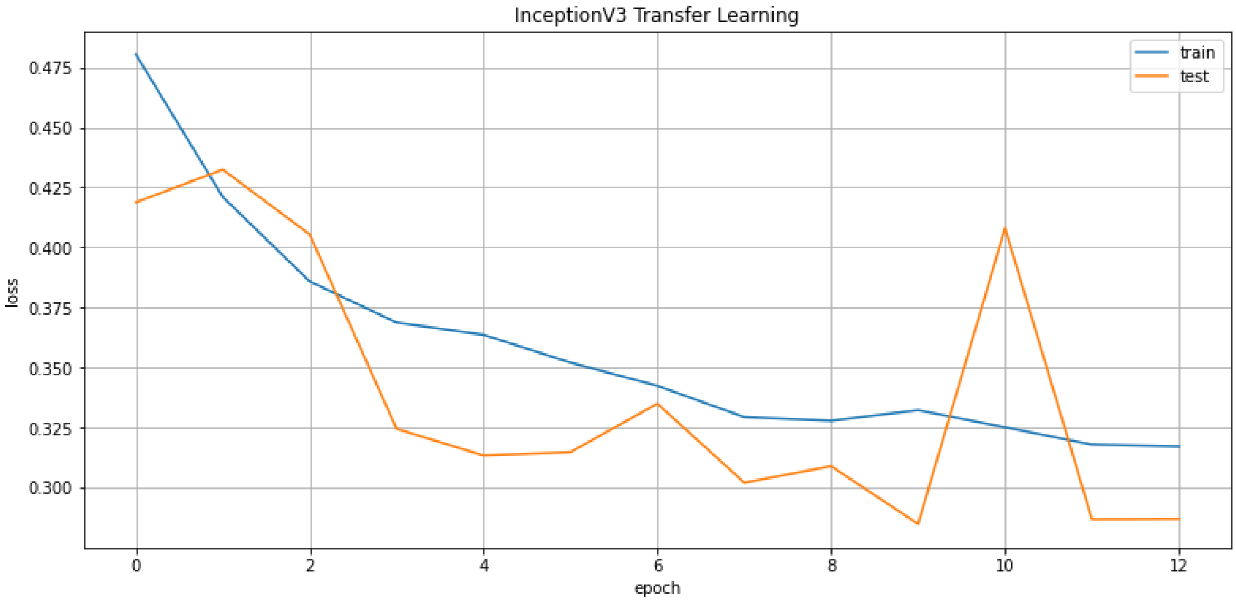


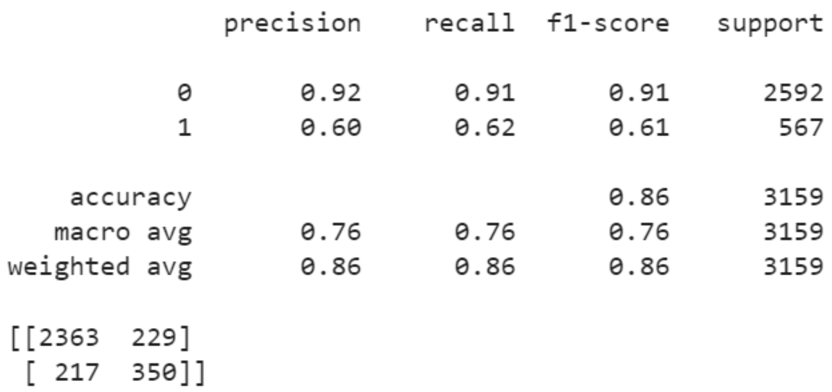
Figure 60 - Train and test set loss for each epoch

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.



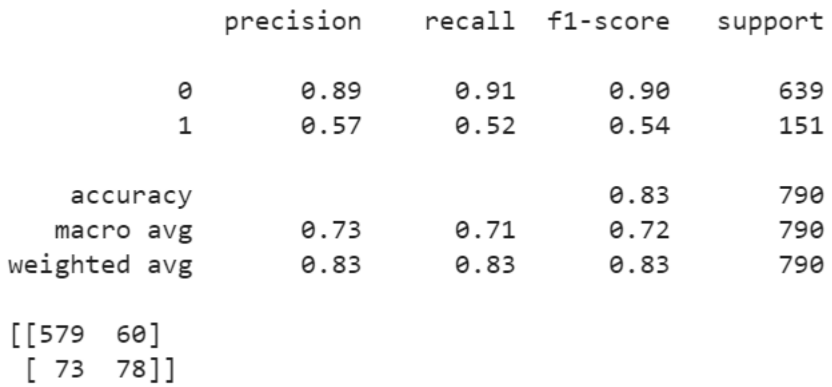
```
              precision    recall  f1-score   support

           0       0.92      0.91      0.91      2592
           1       0.60      0.62      0.61       567

    accuracy                           0.86      3159
   macro avg       0.76      0.76      0.76      3159
weighted avg       0.86      0.86      0.86      3159

[[2363  229]
 [ 217  350]]
```

Figure 61 - Measurements for Inception-V3 on train set



```
              precision    recall  f1-score   support

           0       0.89      0.91      0.90       639
           1       0.57      0.52      0.54       151

    accuracy                           0.83       790
   macro avg       0.73      0.71      0.72       790
weighted avg       0.83      0.83      0.83       790

[[579  60]
 [ 73  78]]
```

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.

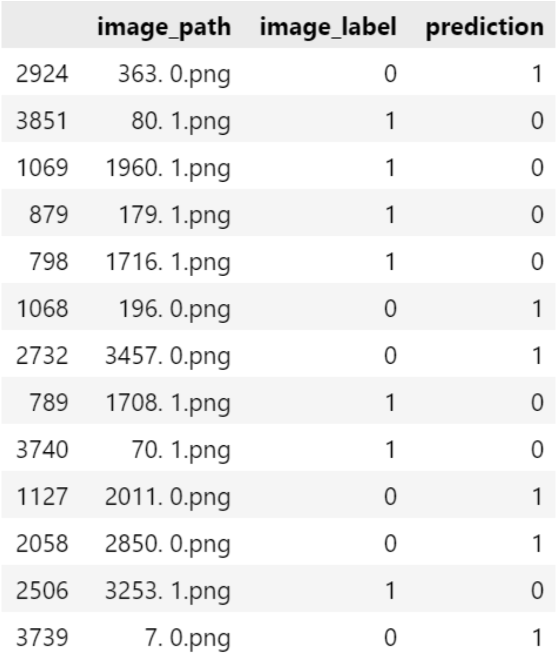
10. Misclassified Images

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
```



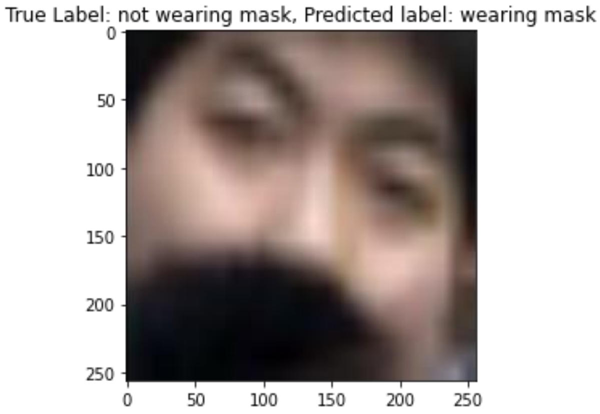
	image_path	image_label	prediction
2924	363. 0.png	0	1
3851	80. 1.png	1	0
1069	1960. 1.png	1	0
879	179. 1.png	1	0
798	1716. 1.png	1	0
1068	196. 0.png	0	1
2732	3457. 0.png	0	1
789	1708. 1.png	1	0
3740	70. 1.png	1	0
1127	2011. 0.png	0	1
2058	2850. 0.png	0	1
2506	3253. 1.png	1	0
3739	7. 0.png	0	1

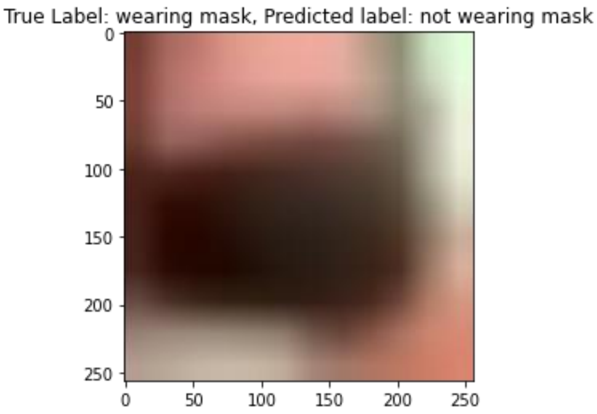
Figure 63 - Misclassified images based on the best model

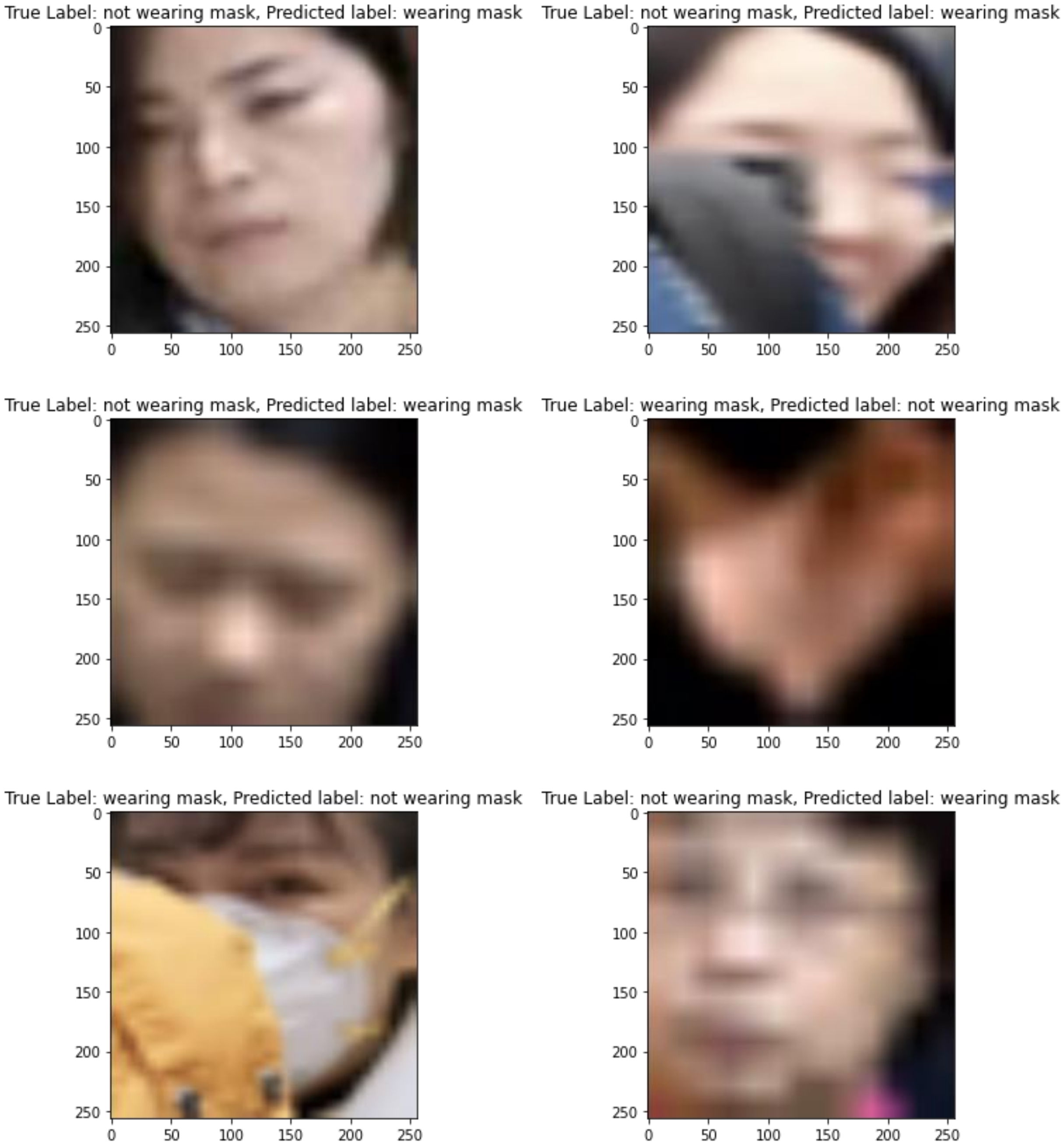
Finally, we can observe the images below:

```
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()
```







True Label: not wearing mask, Predicted label: wearing mask

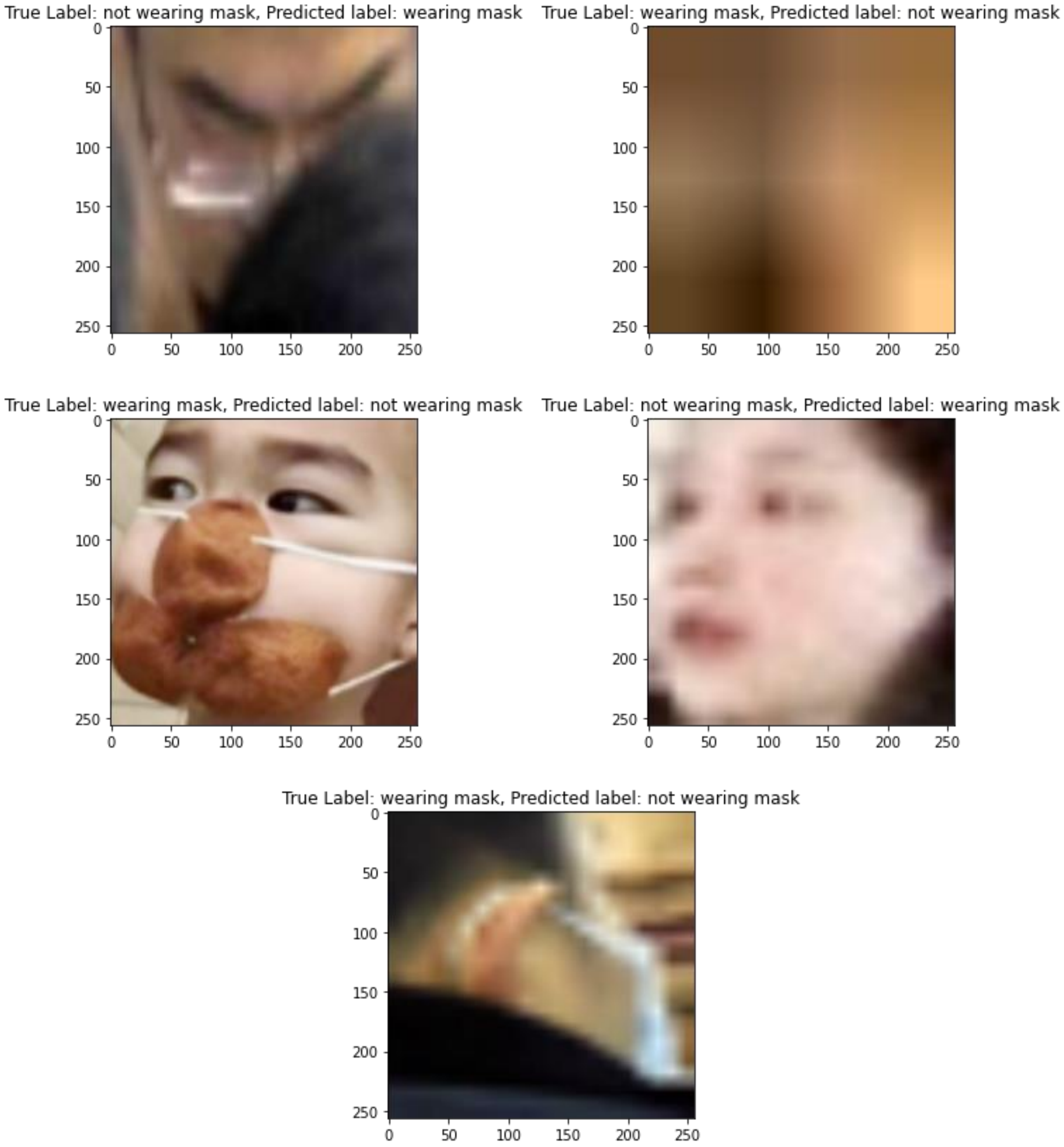
True Label: not wearing mask, Predicted label: wearing mask

True Label: not wearing mask, Predicted label: wearing mask

True Label: wearing mask, Predicted label: not wearing mask

True Label: wearing mask, Predicted label: not wearing mask

True Label: not wearing mask, Predicted label: wearing mask


True Label: not wearing mask, Predicted label: wearing mask

True Label: wearing mask, Predicted label: not wearing mask

True Label: wearing mask, Predicted label: not wearing mask

True Label: not wearing mask, Predicted label: wearing mask

True Label: wearing mask, Predicted label: not wearing mask

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.