DROWSINESS DETECTION

COURSE NAME : EHB 440E - ARTIFICIAL NEURAL NETWORKS

LECTURER : ASSOC. PROF. İSA YILDIRIM

İLKE BURÇAK – 040180755

ÖZLEM YALÇIN – 040170125

ALİ EREN KARADAĞ – 040180218

22/06/2022

Introduction

In the USA, there are about 100,000 crashes, 71,000 injuries, and 1,550 fatalities per year involving drowsy driving as stated by National Safety Council (NSC). And this rate accounts for an estimated 9.5% of all accidents in a year, reported by AAA [1]. Given the multiplicity of these accidents, Drowsiness Detection for drivers is really necessary. Various technologies such as these 3 main detection techniques can be used to detect drowsiness; 1) following the driver's eye aperture and facial features, 2) lane monitoring and vehicle position tracking 3) tracking hand movements on the steering wheel. In this project, we have implemented the first option, which is the fastest method of reacting to drowsiness. There are already new models of top segment cars that use multiple drowsiness detection systems at the same time, and it is expected to become more widespread in the future.

Literature Review

Many methods and approaches have been developed to increase accuracy and speed of Driver Drowsiness Detection. This section attempts to summarize Computer-Vision with Neural Network Model measurements.

To begin with face detection, Viola Jones is the oldest and most popular real-time face recognition algorithm, also used in this project. It consists of the implementation of these 4 critical stages; 1) Calculating Integral Image, 2) Calculating Haar like features, 3) AdaBoost Learning Algorithm, 4) Cascade Filter. Integral Image serves to quickly summarize rectangular regions regardless of filter size[2]. This explains why all cropped frames of Haar Cascade are rectangular. Haar Feature-based Cascade Classifier is an approach based on machine learning and it is pretrained function from many positive and negative images. It uses weak white and black classifiers -called Edge, Line and Four-rectangle Features- just like our convolutional kernel (showed on Fig.1). The Classifier works the same on all face objects such as face, eye, mouth, nose, ear, etc… Then, Adaptive Boosting also known as AdaBoost basically prevents the same Cascade processes from being performed on the entire face and ensures that only the possible region is made, by combining and increasing the weights of the misclassified weak learners in the training set of the model. Finally, Cascade Filter passes results by threshold value to get binary results.

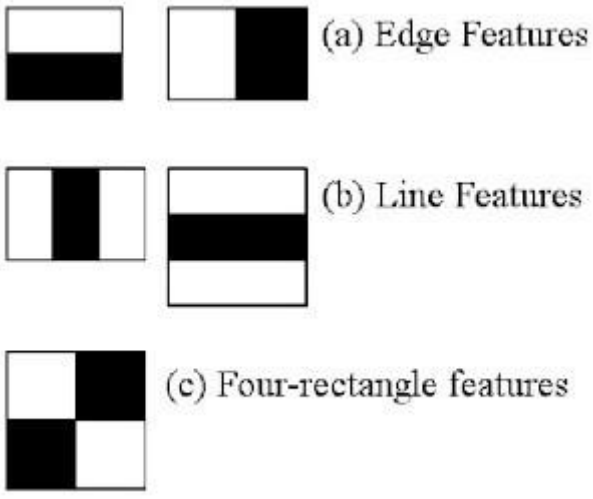


Figure A. Haar Cascade Types

To continue with building an Neural Network, creating your own model is more difficult and longer way also that requires more CPU/GPU. To avoid this problem, you can reuse pre-trained and grand networks by simply changing their last few layers according to your model's demands and this is called Transfer Learning. One of the most well-known Deep Learning (DL) algorithms is the Convolutional Neural Network (CNN), which, when an image is delivered, applies learnable weights and bias to a different place in the image and can thus distinguish between distinct features. Compared to other DL techniques, CNN requires less pre-processing[3]. The visual cortex of the human brain served as inspiration for CNN's design. When input flows through the layers of a CNN, which are made up of several layer types, these layers conduct various processes. In this project, MobileNet, TensorFlow's first mobile computer vision model, was used as the network. MobileNet is built on a simplified design that creates lightweight Deep Neural Networks using depth-wise separable convolutions. When compared to a network with conventional convolutions of the same depth in the nets, MobileNet dramatically reduces the number of parameters. In this project model training and testing predictions using transfer learning are done in the Google Colaboratory. Also called as Colab, is a cloud-based service. For performing machine learning and deep learning operations, Colab is built on the Jupyter Notebook. The Google Colab offers runtime GPU access without charge. For computer vision and other GPU-centric applications, the Google collaborative is helpful. However, since Colab has limited access to computer hardware, webcam tests of the project have been performed on local Jupyter notebook (Anaconda).

Moreover, in this project, 2000 images were used as the training set and 50 images were used as the testing set from the MRL Eye Dataset used as the data set. This dataset includes low- and high-resolution infrared photos that were all taken under varied lightning situations and with various equipment. Testing various features or trainable classifiers is appropriate for this dataset.

Methodology

This section presents a summary of how to fine-tune a pre-trained neural network (MobileNet) that will be utilized to determine whether or not the driver is drowsy later on. The suggested method will detect eye frames in webcam video. Figure 1 depicts an overview of the training and testing approach utilized in the drowsiness detection algorithm.

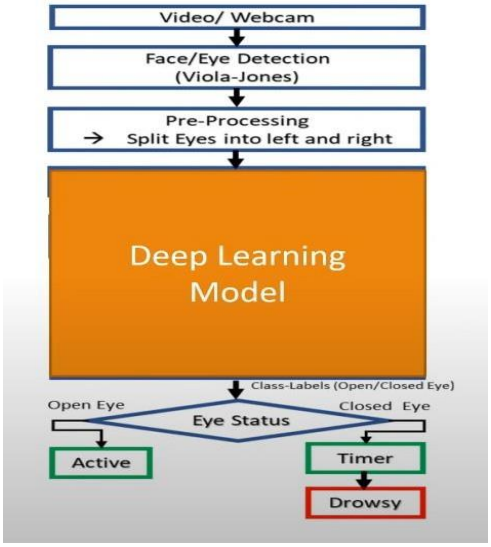


Figure1. An overview of the drowsiness detection process

Proposed method includes five steps.

Step1. Video /webcam

Images required for our network are obtained from real-time videos captured by webcam.

Step2. Face and eye detection by Haar Cascade (Viola-jones) algorithm

2.1. Checking The Network Predictions

First of all using the MRL_Eye Dataset, that is downloaded before, randomly selected image file is read. Then as it is done previously, the colored images turns to the gray ones and adjusted their sizes as 224 x 224.

```
In [19]: img_array = cv2.imread("C:/Users/asus/OneDrive/Desktop/Anaconda3 (64-bit)/Test_Dataset/Closed_Eyes/s0002_00682_0_0_0_0_0_01.png", cv2.IMREAD_GRAYSCALE)
         backtorgb = cv2.resize(backtorgb, (img_size, img_size))
         new_array = cv2.resize(backtorgb, (img_size,img_size))
```

In the next step, since the sizes of the initial images are 86 x 86 and the sizes of the images are now changed to 224 x 224, the array in which we sort the photos is shaped.

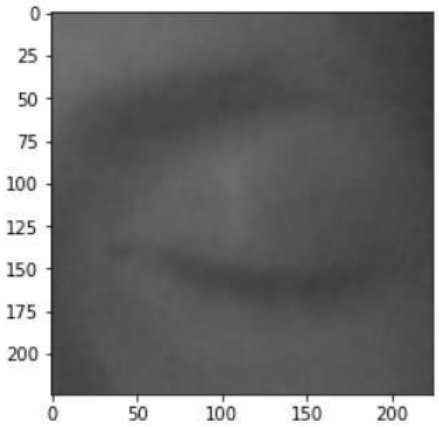
```
In [20]: X_input = np.array(new_array).reshape(1, img_size, img_size, 3)
```

Thus, if dimensions are checked for an only one image, the dimensions can be adjusted as desired. In the output it can be observed that, we got (1, 224, 224, 3).

```
In [21]: X_input.shape
Out[21]: (1, 224, 224, 3)
```

Then, when this image is printed, which is randomly selected from the dataset and edited, below image is obtained.

```
In [22]: plt.imshow(new_array)
Out[22]: <matplotlib.image.AxesImage at 0x11dcd557340>
```



Then image needs to be normalized as follows:

```
In [23]: X_input = X_input/255.0
```

Therefore, to make a prediction, we first assign a variable that we call "prediction". Then, this piece of code enable us to make the prediction when we use the pictures that we normalized our previously saved model as an input:

```
In [24]: prediction = new_model.predict(X_input)

In [25]: prediction
```

It can be observed that, if the photo with eyes open is selected, the model results the value as a positive number, and if a photo with eyes closed is selected, the model results the value as a negative number.

Out [25] : array([[-11.130485]], dtype = float32)

2.2 Creating the Frames for the eyes and Try It on the Different Pictures That Is Not In The Dataset

In the previous section, we enable our algorithm to distinguish between the pictures with eyes open and pictures with eyes closed in a suitable way from the selected dataset. Since our aim is to make the drowsiness detection and giving a decision by only looking at the state of the eyes, in this section in this section we want to make our algorithm to do three thing: successfully identify the eyes, determining their frames, and extracting only the eye part from the picture by selecting a photo that is not in the selected dataset as follows:

First of all, an image is selected and downloaded in a jpg format. Then in order to upload this image and check it, the following code is written:

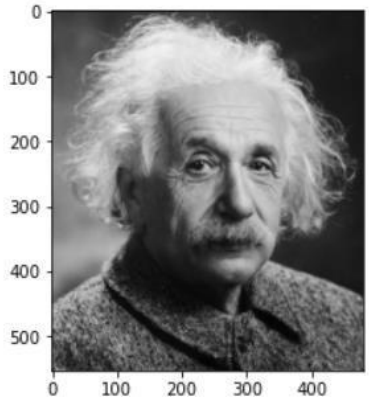
```
In [26]: img = cv2.imread("C:/Users/asus/OneDrive/Desktop/Anaconda3 (64-bit)/einstein2.png")
```

Then in order to observe it, below code is written:

```
In [27]: plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
```

Output:

```
Out[27]: <matplotlib.image.AxesImage at 0x11dcd5c0e20>
```



In order to crop the eyes, the first thing that is needed openCV library. Then the second thing is detecting the eyes and the face and in order to do this, haar cascade algorithm should be used. haar cascade face detection algorithm can be used only by downloading and importing the haarcascade_eye.xml, haarcascade_eye_tree_eyeglasses.xml and haarcascade_frontalface_default.xml files in to the algorithm. To do that following command should be written:

```
In [28]: faceCascade = cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_frontalface_default.xml")
```

```
In [29]: eye_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_eye.xml")
```

Because all the images that we obtained are gray colored images, unknown image should also be converted to the gray color.

```
In [30]: gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Then, different eyes should be detected

```
In [31]: eyes = eye_cascade.detectMultiScale(gray, 1.1, 4)
```

After detected all the eyes, following command going to draw all the eyes and put a green rectangle across it. Note that, in order to set the green color, we simply write (0, 255, 0)

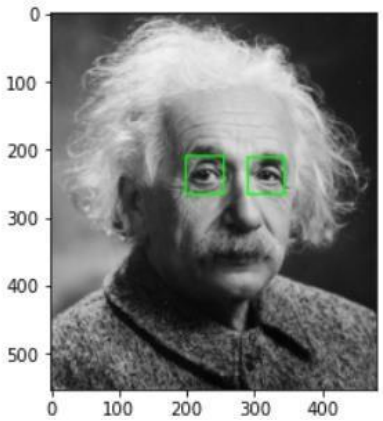
```
In [32]: for(x, y, w, h) in eyes:
             cv2.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2)
```

Now, in order to executed the applied version of the picture where the multiple eyes are determined and the green rectangle is drawn across them can be observed in below:

```
In [33]: plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
```

Output:

```
Out[33]: <matplotlib.image.AxesImage at 0x11da507d790>
```



Next, in order to cropped the eyes following command is used:

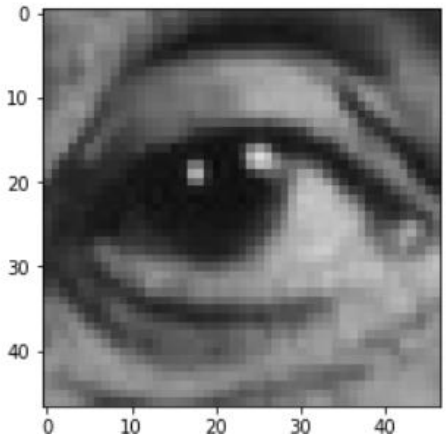
```
In [35]: eye_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_eye.xml")
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
eyes= eye_cascade.detectMultiScale(gray, 1.1,4)

for x,y,w,h in eyes:
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]
    eyess = eye_cascade.detectMultiScale(roi_gray)
    if len(eyess) == 0:
        print("Eyes are not detected!")
    else:
        for(ex,ey,ew,eh) in eyess:
            eyes_roi = roi_color[ey:ey+eh, ex:ex + ew]
```

In order to saw the image of cropped eyes, it is executed as follows:

```
In [36]: plt.imshow(cv2.cvtColor(eyes_roi, cv2.COLOR_BGR2RGB))
```

Output:

```
Out[36]: <matplotlib.image.AxesImage at 0x11da50bf190>
```



In the next step, we wanted to know the dimensions of the image from which the eye was removed, since the dimensions of the photos were previously determined as 224 x 224.

```
In [37]: eyes_roi.shape
Out[37]: (47, 47, 3)
```

Therefore, it is necessary to change the dimensions and normalize the resulting image. So, following command describe this:

```
In [38]: final_image = cv2.resize(eyes_roi, (224,224))
final_image = np.expand_dims(final_image, axis = 0)
final_image = final_image/255.0
```

So, in order to check the new dimensions:

```
In [39]: final_image.shape
Out[39]: (1, 224, 224, 3)
```

Based on the final image obtained, prediction is made in the following command:

```
In [40]: new_model.predict(final_image)
```

By checking the result, a positive number is obtained for the open eye image as it was obtained before.

Out[40] : array([[1.6677274]], dtype = float32)

REALTIME VIDEO DEMO

First of all, we need to add the haar cascade algorithm to our own drowsiness detection algorithm in order to detect whether the eyes are open or closed when the real-time video is opened. Because in order for the photo frames taken with the video in a certain period of time to understand whether the eyes are closed or open after identifying the face, the photo must be scanned with the binary rectangular method, starting from the upper left corner and reaching the lower right corner, as explained before.

```
In [41]: import cv2
         path = "haarcascade_frontalface_default.xml"
         faceCascade = cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_frontalface_default.xml")
```

Then, the following command was written to activate the camera. Then, the following command was written to activate the camera. Index 1 is used when the camera is on, while index 0 is used for when the camera is off. Later, if the camera is not found due to any drive problem, a warning message is printed as following:

```
cap = cv2.VideoCapture(1)

if not cap.isOpened():
    cap = cv2.VideoCapture(0)
if not cap.isOpened():
    raise IOError("Cannot open webcam")
```

In order for the camera to run continuously, take the photo frame and detect the eyes in real-time after finding the face, the following command is written. To determine location the eyes from the video frame, the following part was written first:

```
while True:
    ret,frame = cap.read()
    eye_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_eye.xml")
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    eyes = eye_cascade.detectMultiScale(gray, 1.1,4)
    for x,y,w,h in eyes:
        roi_gray = gray[y:y+h, x:x+h]
        roi_color = frame[y:y+h, x:x+w]
        cv2.rectangle(frame, (x,y), (x+w, y+h), (0, 255, 0), 2)
        eyess = eye_cascade.detectMultiScale(roi_gray)
```

Then the eyes determined from inside the video frame were surrounded by a green rectangle. A warning message is provided if eyes cannot be detected:

```
if len(eyess) == 0:
    print("eyes are not detected")
```

After that, if there is not any problem and eyes are detected successfully, to crop the eyes from the video frame, the following command is written:

```
else:
    for(ex,ey,ew,eh) in eyess:
        eyes_roi = roi_color[ey:ey+eh, ex:ex + ew]
```

Next, the dimensions of the photo are adjusted and normalized for the re-cropped eye frame.

```
final_image = cv2.resize(eyes_roi, (224,224))
final_image = np.expand_dims(final_image, axis = 0)
final_image = final_image/255.0
```

Afterwards, the following command is written to predict whether the cropped eye taken from the video is open or closed in real-time using the saved network model. The prediction event here is done as described before. In other words, if there is a negative result, he gets tired with his eyes closed. Even if a positive number results, the eye is interpreted as open.

```
Predictions = new_model.predict(final_image)
if (Prections>0):
    status = "Open Eyes"
else:
    status = "Closed Eyes"
```

Afterwards, the photo is converted to a gray photo and the following command is typed so that a rectangle around the face can appear in the output.

```
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
print(faceCascade.empty())
faces = faceCascade.detectMultiScale(gray, 1.1, 4)
```

In order to write the comments made on the video on the network model in text format, the following command was written:

```
    #In order to draw a rectangle around the face:
    for (x, y, w, h) in faces:
        cv2.rectangle(frame, (x, y), (x+w, y+w), (0, 255, 0), 2)

    font = cv2.FONT_HERSHEY_SIMPLEX

# To insert the text on the video

    cv2.putText(frame, (50, 50), font, 3, (0, 0, 255), 2, cv2.LINE_4)

    cv2.imshow("Drowsiness Detection Tutorial", frame)
    if cv2.waitKey(2) & 0xFF == ord("q"):
        break

cap.release()
cv2.destroyAllWindows()
```

Step 3. Fine-tuning MobileNet

In the proposed approach, MobileNet is used for transfer learning. MobileNets are small (light weighted) deep neural networks that are well-suited to embedded vision and mobile applications. MobileNets have a simple architecture that employs depthwise separable convolutions. MobileNet makes use of two basic global hyperparameters to effectively balance accuracy and latency. MobileNet has numerous advantages over other cutting-edge convolutional neural networks like VGG16, VGG19, ResNet50, InceptionV3, and Xception. Advantages of MobileNet can be listed as follows:

- Reduced number of parameters
- Reduced network size
- Small, light-weight and low-latency CNN
- Faster performance

Fine-Tuning of MobileNet

```
[ ]  import tensorflow as tf
     import cv2
     import os
     import matplotlib.pyplot as plt
     import numpy as np
     from tensorflow.keras.callbacks import TensorBoard
     import time
```

```
[ ]  NAME = "yeni-{}".format(int(time.time() )  )
```

```
     tensorboard = tf.keras.callbacks.TensorBoard(log_dir ='logs/{}'.format(NAME) )
     #tbCallBack = keras.callbacks.TensorBoard(log_dir='./Graph', histogram_freq=0, write_graph=True, write_images=True)
```

Figure2. code snippet1

Google Colaboratory is utilized for the training and testing of the MobileNet to obtain faster results. Required libraries such as tensorflow , matplotlib , numpy , time and opencv are added as shown above. Tensorboard function is imported and used for the visualization of the network training results.

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

```
     Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mount("/content/gdrive", force_remount=True).
```

```
[ ]  DATADIR = 'gdrive/My Drive/Yeni/Train_Dataset'
     CATEGORIES = ["closed_eye","open_eye"]

     for category in CATEGORIES:
         path = os.path.join(DATADIR,category)
         for img in os.listdir(path):
             img_array = plt.imread(os.path.join(path,img))
             plt.imshow(img_array,cmap = "gray")
             plt.show()
             backtorgb = cv2.cvtColor(img_array,cv2.COLOR_GRAY2RGB)
             break
         break
```

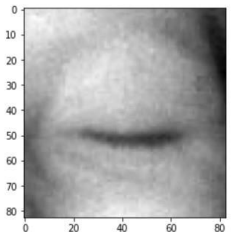


Figure3. code snippet2

MRL eye data set is utilized for the transfer learning of MobileNet. In the original MRL eye data set, there are 2 classes: open eye and close eye each of which has 20.000 images , 40.000 images in total. However, in our project 1000 images are used for each class. A sample image is read from the drive and converted into gray scale in figure2.

```
[ ] img_array.shape

    (83, 83)

[ ] img_size = 224
    new_array = cv2.resize(backtorgb,(img_size,img_size))
    plt.imshow(new_array,cmap = "gray")
    plt.show()
```

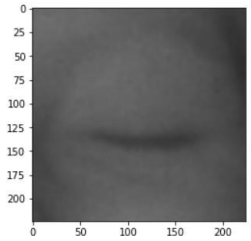


Figure4. code snippet3

Images of MRL eye dataset has a shape of (83,83). Since MobileNet is pre-trained with the images of shape (224,224), MRL dataset images are resized to (224,224) as shown above.

```
[8]  training_data = []

     def create_training_data():
         img_num =0
         for category in CATEGORIES:
             path = os.path.join(DATADIR,category)
             class_num = CATEGORIES.index(category)
             for img in os.listdir(path):
                 try:
                   img_array = plt.imread(os.path.join(path,img))
                   backtorgb = cv2.cvtColor(img_array,cv2.COLOR_GRAY2RGB)
                   new_array = cv2.resize(backtorgb, (img_size,img_size))
                   training_data.append([new_array,class_num])
                   img_num+=1
                   if img_num == 1000:
                     img_num=0
                     break
                 except Exception as e:
                     pass

[9]  create_training_data()
```

Figure5. code snippet4

Firstly, training_data[] array is created to store image data and their corresponding labels. “create_training_data” function is written in order to label each image according to their classes and then store them into training_data array. The two classes “close eye” and “open eye” are labeled as “0” and “1”. As aforementioned above , images are read from the drive path into an array called “img_array”. Since MobileNet is pre-trained with RGB images, all the gray scale images are converted into RGB then, resized to (224,224). 1000 images are used for each classes.

```
[ ] import random
    random.shuffle(training_data)

[ ] X = []
    y = []

    for features, label in training_data:
        X.append(features)
        y.append(label)

    X = np.array(X).reshape(-1,img_size,img_size,3)

[ ] X = X/ 255.0
    y =np.array(y)

    import pickle
    pickle_out = open("X.data","wb")
    pickle.dump(X,pickle_out)
    pickle_out.close()

    pickle_out = open("y.labels","wb")
    pickle.dump(y,pickle_out)
    pickle_out.close()

[ ] import tensorflow as tf
    from tensorflow import keras
    from tensorflow.keras import layers
```

Figure6. code snippet5

Training data are shuffled randomly before training the MobileNet. Image data and their corresponding labels are converted into numpy arrays X and y, respectively. Values of X array including image data range from 0 to 255. In order to make the proposed algorithm computationally effective, each element of the X numpy array is normalized by dividing by 255. Using pickle library , image data and labels are stored for later use. Required packages such as keras and layers are imported for adding new layers to the MobileNet.

```
[ ] model = tf.keras.applications.mobilenet.MobileNet()

[ ] model.summary()

[ ] base_input = model.layers[0].input

[ ] base_output = model.layers[-4].output

[ ] Flat_layer = layers.Flatten()(base_output)
    final_output1 = layers.Dense(64)(Flat_layer)
    final_output2 = layers.Activation("relu")(final_output1)
    final_output3 = layers.Dense(1)(final_output2)
    final_output = layers.Activation("sigmoid")(final_output3)

[ ] new_model = keras.Model(inputs =base_input, outputs = final_output)

[ ] new_model.summary()

[ ] opt = keras.optimizers.Adam(learning_rate=0.0001)
    new_model.compile(loss = "binary_crossentropy",optimizer = opt,metrics = ["accuracy"])
```

Figure7. code snippet6 (**Best case of the fine-tuning trails**)

MobileNet is downloaded by the command tf.keras.applications.mobilenet.MobileNet(). First layer of the MobileNet is also used as input layer in the new_model. The fourth-to-last layer is selected as the beginning of the output layer. The fourth-to-last layer is a dropout layer which has a shape of (1, 1, 1024). Thus, a flatten layer is added to make the dropout layer one dimensional (1024). Then the output of the flatten layer is given to a dense layer which has 64 neurons. After the dense layer, an activation layer which has a “RELU” function as activation function is added. One significant advantage of RELU is that the gradient is less likely to vanish. Learning is accelerated by the constant gradient of ReLUs. Finally, a dense layer which has one neuron is added , it has only one neuron because there are 2 classes: closed eye (“0”) and open eye (“1”). Then, an activation layer which has a “sigmoid” function as activation function is added. It's one of the most effective

normalized functions available. It provides a clear forecast using 1 and 0. Another advantage of sigmoid is that it returns a result in the range of (0,1) when used with (- infinite, + infinite) as in the linear function .After the fine-tuning process, new_model is created. Last layers of the MobileNet and the new_model is given below. This fine-tuned model gave the best results in terms of accuracy and loss.

conv_dw_13_relu (ReLU)	(None, 7, 7, 1024)	0
conv_pw_13 (Conv2D)	(None, 7, 7, 1024)	1048576
conv_pw_13_bn (BatchNormalization)	(None, 7, 7, 1024)	4096
conv_pw_13_relu (ReLU)	(None, 7, 7, 1024)	0
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1, 1, 1024)	0
dropout (Dropout)	(None, 1, 1, 1024)	0
conv_preds (Conv2D)	(None, 1, 1, 1000)	1025000
reshape_2 (Reshape)	(None, 1000)	0
predictions (Activation)	(None, 1000)	0

Total params: 4,253,864
Trainable params: 4,231,976
Non-trainable params: 21,888

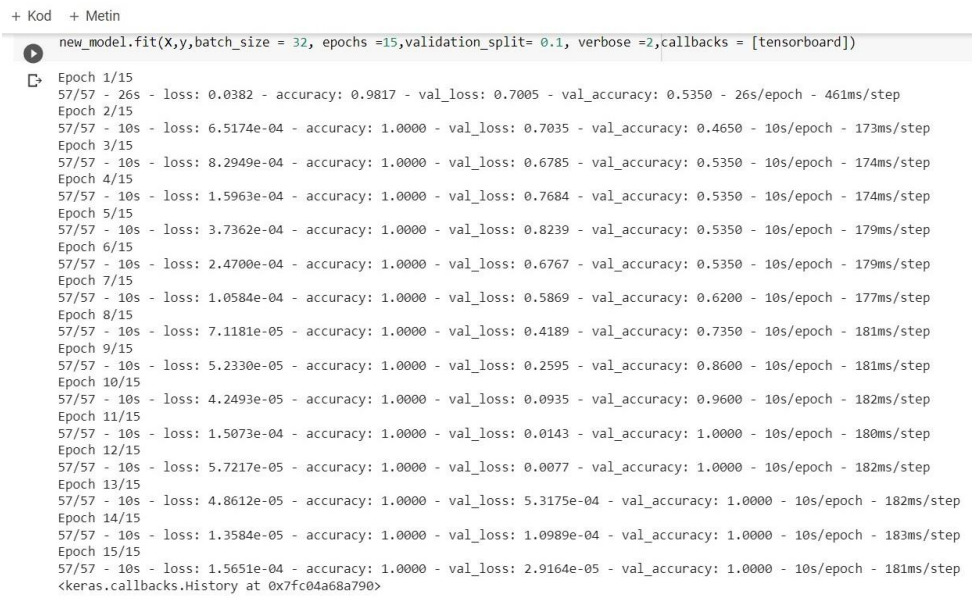
conv_dw_13_bn (BatchNormalization)	(None, 7, 7, 1024)	4096
conv_dw_13_relu (ReLU)	(None, 7, 7, 1024)	0
conv_pw_13 (Conv2D)	(None, 7, 7, 1024)	1048576
conv_pw_13_bn (BatchNormalization)	(None, 7, 7, 1024)	4096
conv_pw_13_relu (ReLU)	(None, 7, 7, 1024)	0
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1, 1, 1024)	0
dropout (Dropout)	(None, 1, 1, 1024)	0
flatten (Flatten)	(None, 1024)	0
dense (Dense)	(None, 64)	65600
activation (Activation)	(None, 64)	0
dense_1 (Dense)	(None, 1)	65
activation_1 (Activation)	(None, 1)	0

Total params: 3,294,529
Trainable params: 3,272,641
Non-trainable params: 21,888

Figure8. Last layers of the MobileNet and new_model

For the optimizer , Adam is chosen and after several testing, learning rate of 10^{-4} is found to give more reliable results. Since we have 2 classes, binary cross entropy is chosen as loss function. Our model performance is evaluated based on accuracy metric.

Results of the best case of the fine-tuning trails



```
+ Kod  + Metin
new_model.fit(X,y,batch_size = 32, epochs =15,validation_split= 0.1, verbose =2,callbacks = [tensorboard])

Epoch 1/15
57/57 - 26s - loss: 0.0382 - accuracy: 0.9817 - val_loss: 0.7005 - val_accuracy: 0.5350 - 26s/epoch - 461ms/step
Epoch 2/15
57/57 - 10s - loss: 6.5174e-04 - accuracy: 1.0000 - val_loss: 0.7035 - val_accuracy: 0.4650 - 10s/epoch - 173ms/step
Epoch 3/15
57/57 - 10s - loss: 8.2949e-04 - accuracy: 1.0000 - val_loss: 0.6785 - val_accuracy: 0.5350 - 10s/epoch - 174ms/step
Epoch 4/15
57/57 - 10s - loss: 1.5963e-04 - accuracy: 1.0000 - val_loss: 0.7684 - val_accuracy: 0.5350 - 10s/epoch - 174ms/step
Epoch 5/15
57/57 - 10s - loss: 3.7362e-04 - accuracy: 1.0000 - val_loss: 0.8239 - val_accuracy: 0.5350 - 10s/epoch - 179ms/step
Epoch 6/15
57/57 - 10s - loss: 2.4700e-04 - accuracy: 1.0000 - val_loss: 0.6767 - val_accuracy: 0.5350 - 10s/epoch - 179ms/step
Epoch 7/15
57/57 - 10s - loss: 1.0584e-04 - accuracy: 1.0000 - val_loss: 0.5869 - val_accuracy: 0.6200 - 10s/epoch - 177ms/step
Epoch 8/15
57/57 - 10s - loss: 7.1181e-05 - accuracy: 1.0000 - val_loss: 0.4189 - val_accuracy: 0.7350 - 10s/epoch - 181ms/step
Epoch 9/15
57/57 - 10s - loss: 5.2330e-05 - accuracy: 1.0000 - val_loss: 0.2595 - val_accuracy: 0.8600 - 10s/epoch - 181ms/step
Epoch 10/15
57/57 - 10s - loss: 4.2493e-05 - accuracy: 1.0000 - val_loss: 0.0935 - val_accuracy: 0.9600 - 10s/epoch - 182ms/step
Epoch 11/15
57/57 - 10s - loss: 1.5073e-04 - accuracy: 1.0000 - val_loss: 0.0143 - val_accuracy: 1.0000 - 10s/epoch - 180ms/step
Epoch 12/15
57/57 - 10s - loss: 5.7217e-05 - accuracy: 1.0000 - val_loss: 0.0077 - val_accuracy: 1.0000 - 10s/epoch - 182ms/step
Epoch 13/15
57/57 - 10s - loss: 4.8612e-05 - accuracy: 1.0000 - val_loss: 5.3175e-04 - val_accuracy: 1.0000 - 10s/epoch - 182ms/step
Epoch 14/15
57/57 - 10s - loss: 1.3584e-05 - accuracy: 1.0000 - val_loss: 1.0989e-04 - val_accuracy: 1.0000 - 10s/epoch - 183ms/step
Epoch 15/15
57/57 - 10s - loss: 1.5651e-04 - accuracy: 1.0000 - val_loss: 2.9164e-05 - val_accuracy: 1.0000 - 10s/epoch - 181ms/step
<keras.callbacks.History at 0x7fc04a68a790>
```

Figure9. Results of the best case

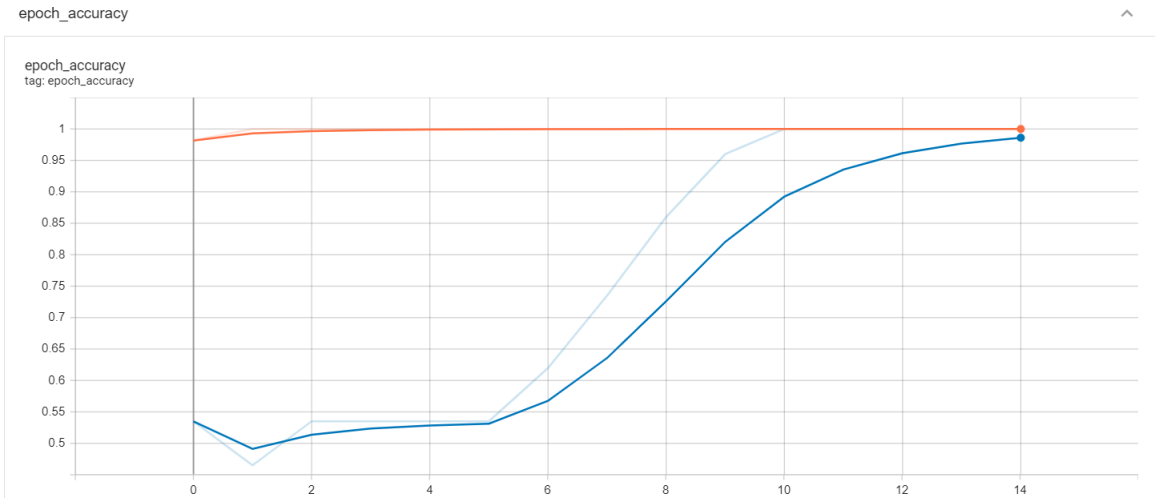


Figure10. Epoch accuracy

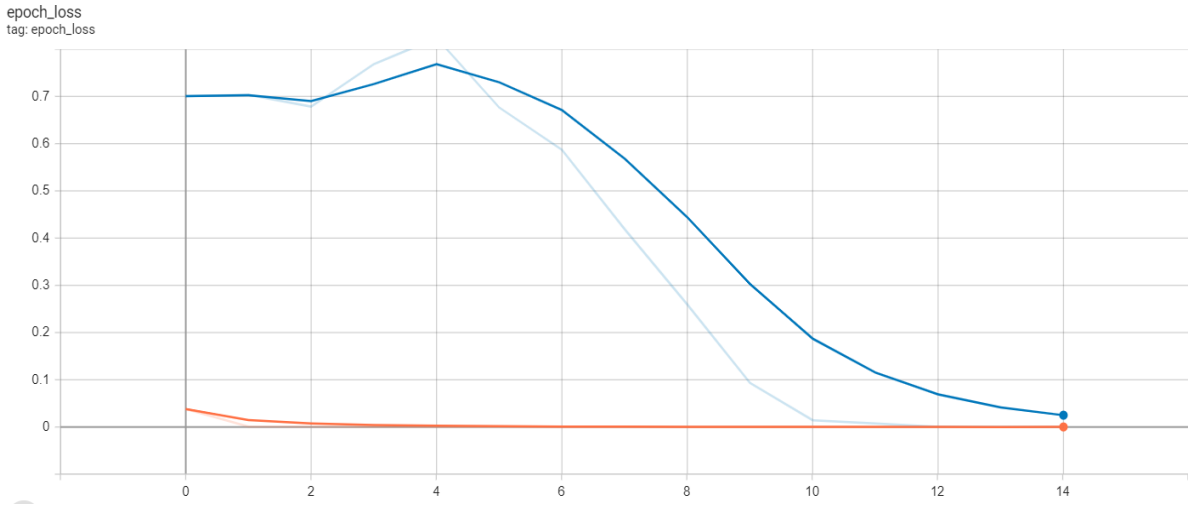


Figure11. Epoch loss

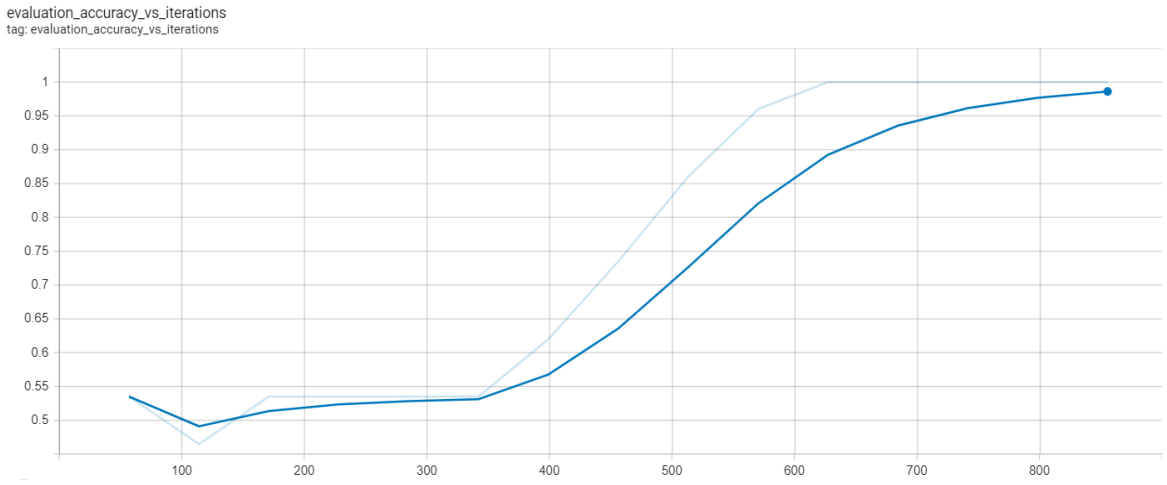


Figure12. Evaluation accuracy versus iterations

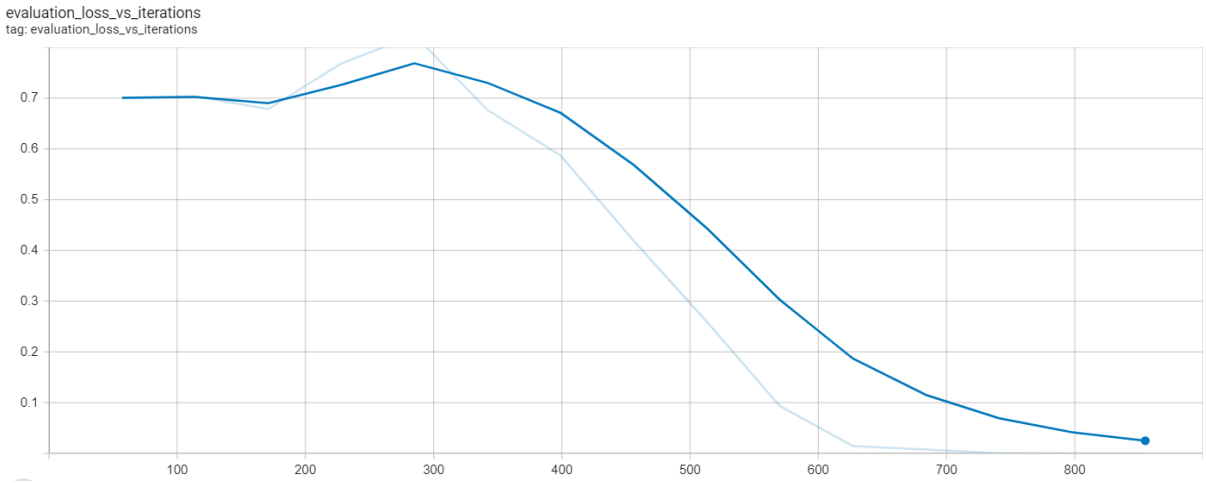


Figure13. Evaluation loss versus iterations

In the four graphs above, orange colored lines indicate train set results, blue colored lines indicate validation set results.

Trials of the Dataset Training

Transfer learning is a method based on heuristic approach . Approximately, seven different fine-tuning models are trained and the best one among them is selected and combined with the Haar Cascade algorithm.

Fine-Tuning Trail 1

```
x = mobile.layers[-5].output # layers up to fifth-to-last layer is selected
Flat_layer = layers.Flatten()(x)#last layer is made 1D with flatten layer
final_output = layers.Dense(1)(Flat_layer)
final_output = layers.Activation("sigmoid")(final_output)
for layer in model.layers[:-23]:
   layer.trainable = False
```

```
model.fit(X,y,epochs =10,validation_split= 0.1, verbose =2)

Epoch 1/10
113/113 - 204s - loss: 0.5709 - accuracy: 0.6753 - val_loss: 2.8413 - val_accuracy: 0.4950 - 204s/epoch - 2s/step
Epoch 2/10
113/113 - 200s - loss: 0.3316 - accuracy: 0.8550 - val_loss: 5.7239 - val_accuracy: 0.5050 - 200s/epoch - 2s/step
Epoch 3/10
113/113 - 201s - loss: 0.2991 - accuracy: 0.8747 - val_loss: 2.6235 - val_accuracy: 0.4950 - 201s/epoch - 2s/step
Epoch 4/10
113/113 - 200s - loss: 0.2837 - accuracy: 0.8869 - val_loss: 6.7171 - val_accuracy: 0.5050 - 200s/epoch - 2s/step
Epoch 5/10
113/113 - 200s - loss: 0.2440 - accuracy: 0.9092 - val_loss: 3.0746 - val_accuracy: 0.5050 - 200s/epoch - 2s/step
Epoch 6/10
113/113 - 199s - loss: 0.2874 - accuracy: 0.8856 - val_loss: 2.3335 - val_accuracy: 0.5050 - 199s/epoch - 2s/step
Epoch 7/10
113/113 - 201s - loss: 0.2231 - accuracy: 0.9147 - val_loss: 9.2489 - val_accuracy: 0.5050 - 201s/epoch - 2s/step
Epoch 8/10
113/113 - 200s - loss: 0.2027 - accuracy: 0.9192 - val_loss: 14.5191 - val_accuracy: 0.5050 - 200s/epoch - 2s/step
Epoch 9/10
113/113 - 200s - loss: 0.3809 - accuracy: 0.8131 - val_loss: 4.2729 - val_accuracy: 0.5050 - 200s/epoch - 2s/step
Epoch 10/10
113/113 - 198s - loss: 0.3113 - accuracy: 0.8672 - val_loss: 3.0342 - val_accuracy: 0.5050 - 198s/epoch - 2s/step
<keras.callbacks.History at 0x7f36b7ad41d0>
```

Figure14. Result of the fine-tuning trail 1

As shown in the code above, the fifth-to-last layer is selected as the beginning of the output layer. Layers up to the twenty-third-to-last layer are made frozen so that weights of these layers are made non-trainable.

Fine-Tuning Trail 2 (no freezing of layers)

```
x = mobile.layers[-5].output
Flat_layer = layers.Flatten()(x)

final_output1 = layers.Dense(64)(Flat_layer)
output1 = layers.Activation("relu")(final_output1)
final_output2 = layers.Dense(1)(output1)
output_last = layers.Activation("sigmoid")(final_output2)
model = Model(inputs = mobile.input, outputs = output_last )
```

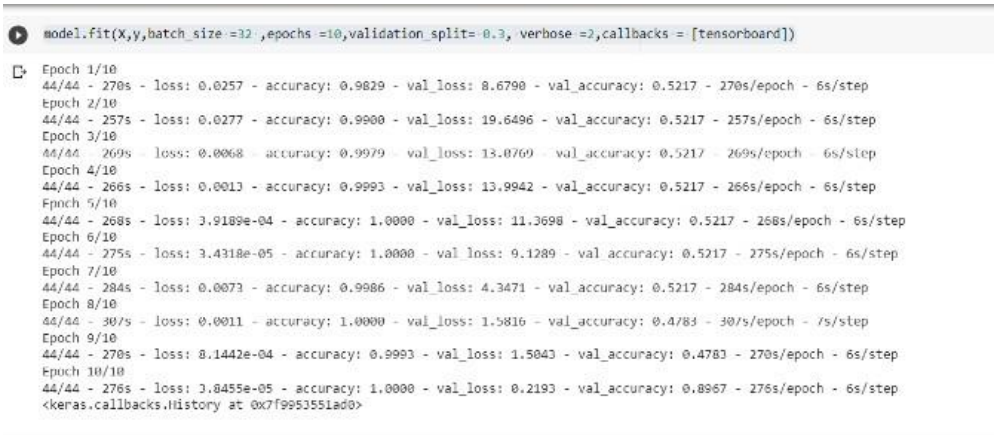


Figure15. Result of the fine-tuning trail 2

Fine-Tuning Trail 3

```
base_input = model.layers[0].input
base_output = model.layers[-4].output
Flat_layer = layers.Flatten()(base_output)
final_output = layers.Dense(1)(Flat_layer)
final_output = layers.Activation("sigmoid")(final_output)
new_model = keras.Model(inputs =base_input, outputs = final_output)
```

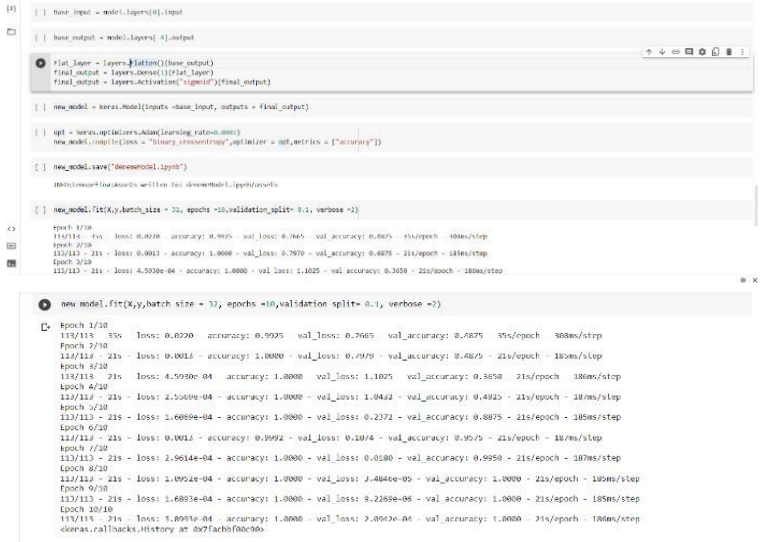


Figure16. Result of the fine-tuning trail 3

Fine-Tuning Trail 4 (10 and 15 epochs): 4000 images from MRL data set are used in total

```
[18] base_input = model.layers[0].input

[19] base_output = model.layers[-4].output

[20] Flat_layer = layers.Flatten()(base_output)
     final_output1 = layers.Dense(64)(Flat_layer)
     final_output2 = layers.Activation("relu")(final_output1)
     final_output3 = layers.Dense(1)(final_output2)
     final_output = layers.Activation("sigmoid")(final_output3)

[21] new_model = keras.Model(inputs =base_input, outputs = final_output)

[22] opt = keras.optimizers.Adam(learning_rate=0.0001)
     new_model.compile(loss = "binary_crossentropy",optimizer = opt,metrics = ["accuracy"])

     new_model.save("denemeModel.ipynb")

[23] new_model.fit(X,y,batch_size = 32, epochs =10,validation_split= 0.1, verbose =2,callbacks = [tensorboard])

     Epoch 1/10
     113/113 - 36s - loss: 0.0263 - accuracy: 0.9889 - val_loss: 0.6932 - val_accuracy: 0.5150 - 36s/epoch - 319ms/step
     Epoch 2/10
     113/113 - 21s - loss: 0.0013 - accuracy: 0.9997 - val_loss: 0.6719 - val_accuracy: 0.5150 - 21s/epoch - 184ms/step
     Epoch 3/10
     113/113 - 21s - loss: 0.0057 - accuracy: 0.9978 - val_loss: 1.6659 - val_accuracy: 0.4850 - 21s/epoch - 189ms/step
     Epoch 4/10
     113/113 - 22s - loss: 6.2478e-04 - accuracy: 1.0000 - val_loss: 0.6950 - val_accuracy: 0.6725 - 22s/epoch - 191ms/step
```

```
new_model.fit(X,y,batch_size = 32, epochs =10,validation_split= 0.1, verbose =2,callbacks = [tensorboard])

Epoch 1/10
113/113 - 36s - loss: 0.0263 - accuracy: 0.9889 - val_loss: 0.6932 - val_accuracy: 0.5150 - 36s/epoch - 319ms/step
Epoch 2/10
113/113 - 21s - loss: 0.0013 - accuracy: 0.9997 - val_loss: 0.6719 - val_accuracy: 0.5150 - 21s/epoch - 184ms/step
Epoch 3/10
113/113 - 21s - loss: 0.0057 - accuracy: 0.9978 - val_loss: 1.6659 - val_accuracy: 0.4850 - 21s/epoch - 189ms/step
Epoch 4/10
113/113 - 22s - loss: 6.2478e-04 - accuracy: 1.0000 - val_loss: 0.6950 - val_accuracy: 0.6725 - 22s/epoch - 191ms/step
Epoch 5/10
113/113 - 21s - loss: 2.8487e-04 - accuracy: 1.0000 - val_loss: 1.2415 - val_accuracy: 0.6925 - 21s/epoch - 188ms/step
Epoch 6/10
113/113 - 21s - loss: 0.0013 - accuracy: 0.9994 - val_loss: 0.0276 - val_accuracy: 0.9925 - 21s/epoch - 185ms/step
Epoch 7/10
113/113 - 21s - loss: 9.2191e-05 - accuracy: 1.0000 - val_loss: 0.0128 - val_accuracy: 0.9975 - 21s/epoch - 189ms/step
Epoch 8/10
113/113 - 21s - loss: 6.1431e-05 - accuracy: 1.0000 - val_loss: 0.0072 - val_accuracy: 0.9975 - 21s/epoch - 187ms/step
Epoch 9/10
113/113 - 21s - loss: 3.7029e-05 - accuracy: 1.0000 - val_loss: 0.0056 - val_accuracy: 0.9975 - 21s/epoch - 188ms/step
Epoch 10/10
113/113 - 21s - loss: 7.2435e-05 - accuracy: 1.0000 - val_loss: 0.0058 - val_accuracy: 0.9975 - 21s/epoch - 186ms/step
<keras.callbacks.History at 0x7f03366b8b50>
```

```
new_model.fit(X,y,batch_size = 32, epochs =15,validation_split= 0.1, verbose =2,callbacks = [tensorboard])

Epoch 1/15
57/57 - 27s - loss: 0.0327 - accuracy: 0.9867 - val_loss: 1.2918 - val_accuracy: 0.4650 - 27s/epoch - 466ms/step
Epoch 2/15
57/57 - 10s - loss: 5.0157e-04 - accuracy: 1.0000 - val_loss: 1.8713 - val_accuracy: 0.4650 - 10s/epoch - 174ms/step
Epoch 3/15
57/57 - 10s - loss: 0.0027 - accuracy: 0.9994 - val_loss: 0.8958 - val_accuracy: 0.4650 - 10s/epoch - 174ms/step
Epoch 4/15
57/57 - 10s - loss: 0.0031 - accuracy: 0.9994 - val_loss: 0.8333 - val_accuracy: 0.5350 - 10s/epoch - 176ms/step
Epoch 5/15
57/57 - 10s - loss: 2.4711e-04 - accuracy: 1.0000 - val_loss: 0.9420 - val_accuracy: 0.5350 - 10s/epoch - 178ms/step
Epoch 6/15
57/57 - 10s - loss: 2.7257e-04 - accuracy: 1.0000 - val_loss: 0.9584 - val_accuracy: 0.4750 - 10s/epoch - 179ms/step
Epoch 7/15
57/57 - 10s - loss: 1.1498e-04 - accuracy: 1.0000 - val_loss: 1.1238 - val_accuracy: 0.4650 - 10s/epoch - 178ms/step
Epoch 8/15
57/57 - 10s - loss: 1.1243e-04 - accuracy: 1.0000 - val_loss: 0.6953 - val_accuracy: 0.5950 - 10s/epoch - 177ms/step
Epoch 9/15
57/57 - 10s - loss: 8.8393e-05 - accuracy: 1.0000 - val_loss: 0.2136 - val_accuracy: 0.8950 - 10s/epoch - 176ms/step
Epoch 10/15
57/57 - 10s - loss: 6.2262e-05 - accuracy: 1.0000 - val_loss: 0.0867 - val_accuracy: 0.9800 - 10s/epoch - 179ms/step
Epoch 11/15
57/57 - 10s - loss: 2.6487e-05 - accuracy: 1.0000 - val_loss: 0.0336 - val_accuracy: 0.9900 - 10s/epoch - 178ms/step
Epoch 12/15
57/57 - 10s - loss: 3.8203e-05 - accuracy: 1.0000 - val_loss: 0.0129 - val_accuracy: 1.0000 - 10s/epoch - 183ms/step
Epoch 13/15
57/57 - 10s - loss: 2.7589e-05 - accuracy: 1.0000 - val_loss: 0.0027 - val_accuracy: 1.0000 - 10s/epoch - 180ms/step
Epoch 14/15
57/57 - 10s - loss: 1.9191e-05 - accuracy: 1.0000 - val_loss: 5.1200e-04 - val_accuracy: 1.0000 - 10s/epoch - 180ms/step
Epoch 15/15
57/57 - 10s - loss: 6.1550e-05 - accuracy: 1.0000 - val_loss: 7.0252e-05 - val_accuracy: 1.0000 - 10s/epoch - 182ms/step
<keras.callbacks.History at 0x7f58565eef10>
```

Figure17. Result of the fine-tuning trail 4

Fine-tuning Trail 5 (same model but a different data set is used for training,2000 images)

```
new_model.fit(X,y,batch_size = 32, epochs =15,validation_split= 0.1, verbose =2,callbacks = [tensorboard])

Epoch 1/15
57/57 - 321s - loss: 0.0737 - accuracy: 0.9739 - val_loss: 1.0675 - val_accuracy: 0.4950 - 321s/epoch - 6s/step
Epoch 2/15
57/57 - 300s - loss: 0.0086 - accuracy: 0.9972 - val_loss: 0.7080 - val_accuracy: 0.4950 - 300s/epoch - 5s/step
Epoch 3/15
57/57 - 308s - loss: 0.0089 - accuracy: 0.9967 - val_loss: 0.9962 - val_accuracy: 0.4950 - 308s/epoch - 5s/step
Epoch 4/15
57/57 - 296s - loss: 0.0056 - accuracy: 0.9978 - val_loss: 2.1036 - val_accuracy: 0.4950 - 296s/epoch - 5s/step
Epoch 5/15
57/57 - 294s - loss: 0.0020 - accuracy: 0.9994 - val_loss: 2.1764 - val_accuracy: 0.4950 - 294s/epoch - 5s/step
Epoch 6/15
57/57 - 298s - loss: 0.0029 - accuracy: 0.9989 - val_loss: 2.3923 - val_accuracy: 0.4950 - 298s/epoch - 5s/step
Epoch 7/15
57/57 - 300s - loss: 3.9802e-04 - accuracy: 1.0000 - val_loss: 2.0270 - val_accuracy: 0.4950 - 300s/epoch - 5s/step
Epoch 8/15
57/57 - 298s - loss: 2.6986e-04 - accuracy: 1.0000 - val_loss: 1.4444 - val_accuracy: 0.5750 - 298s/epoch - 5s/step
Epoch 9/15
57/57 - 299s - loss: 8.7400e-05 - accuracy: 1.0000 - val_loss: 0.7412 - val_accuracy: 0.7250 - 299s/epoch - 5s/step
Epoch 10/15
57/57 - 299s - loss: 5.2664e-04 - accuracy: 1.0000 - val_loss: 0.3610 - val_accuracy: 0.8600 - 299s/epoch - 5s/step
Epoch 11/15
57/57 - 296s - loss: 4.0330e-04 - accuracy: 1.0000 - val_loss: 0.2151 - val_accuracy: 0.9300 - 296s/epoch - 5s/step
Epoch 12/15
57/57 - 300s - loss: 1.0459e-04 - accuracy: 1.0000 - val_loss: 0.1151 - val_accuracy: 0.9650 - 300s/epoch - 5s/step
Epoch 13/15
57/57 - 301s - loss: 0.0031 - accuracy: 0.9994 - val_loss: 0.0846 - val_accuracy: 0.9550 - 301s/epoch - 5s/step
Epoch 14/15
57/57 - 298s - loss: 0.0311 - accuracy: 0.9922 - val_loss: 0.2128 - val_accuracy: 0.9500 - 298s/epoch - 5s/step
Epoch 15/15
57/57 - 298s - loss: 0.0031 - accuracy: 0.9983 - val_loss: 0.0998 - val_accuracy: 0.9700 - 298s/epoch - 5s/step
<keras.callbacks.History at 0x7f3de423b7d0>
```

Figure18. Result of the fine-tuning trail 5

Fine-tuning Trail 6

```
base_input = model.layers[0].input
base_output = model.layers[-4].output
Flat_layer = layers.Flatten()(base_output)
final_output3 = layers.Dense(1)(Flat_layer)
final_output = layers.Activation("sigmoid")(final_output3)
```

```
new_model.fit(X,y,batch_size = 32, epochs =10,validation_split= 0.2, verbose =2,callbacks = [tensorboard])

Epoch 1/10
50/50 - 13s - loss: 0.0360 - accuracy: 0.9819 - val_loss: 3.0682 - val_accuracy: 0.4750 - 13s/epoch - 261ms/step
Epoch 2/10
50/50 - 9s - loss: 7.0107e-04 - accuracy: 1.0000 - val_loss: 3.2777 - val_accuracy: 0.4750 - 9s/epoch - 189ms/step
Epoch 3/10
50/50 - 10s - loss: 0.0014 - accuracy: 0.9994 - val_loss: 1.9123 - val_accuracy: 0.5600 - 10s/epoch - 191ms/step
Epoch 4/10
50/50 - 10s - loss: 6.4350e-04 - accuracy: 1.0000 - val_loss: 0.5675 - val_accuracy: 0.7775 - 10s/epoch - 191ms/step
Epoch 5/10
50/50 - 10s - loss: 1.6112e-04 - accuracy: 1.0000 - val_loss: 0.0121 - val_accuracy: 1.0000 - 10s/epoch - 191ms/step
Epoch 6/10
50/50 - 10s - loss: 2.5486e-04 - accuracy: 1.0000 - val_loss: 0.0012 - val_accuracy: 1.0000 - 10s/epoch - 191ms/step
Epoch 7/10
50/50 - 10s - loss: 1.6076e-04 - accuracy: 1.0000 - val_loss: 1.5824e-05 - val_accuracy: 1.0000 - 10s/epoch - 193ms/step
Epoch 8/10
50/50 - 10s - loss: 2.3068e-05 - accuracy: 1.0000 - val_loss: 4.0111e-06 - val_accuracy: 1.0000 - 10s/epoch - 190ms/step
Epoch 9/10
50/50 - 10s - loss: 2.8119e-05 - accuracy: 1.0000 - val_loss: 2.9599e-06 - val_accuracy: 1.0000 - 10s/epoch - 195ms/step
Epoch 10/10
50/50 - 10s - loss: 4.7666e-05 - accuracy: 1.0000 - val_loss: 2.7329e-06 - val_accuracy: 1.0000 - 10s/epoch - 192ms/step
<keras.callbacks.History at 0x7fb8791fc2d0>
```

Figure19. Result of the fine-tuning trail 6

Fine-tuning Trail 7 (4000 images are used for training)

```
Flat_layer = layers.Flatten()(base_output)
final_output3 = layers.Dense(1)(Flat_layer)
final_output = layers.Activation("sigmoid")(final_output3)
```

```
new_model.fit(X,y,batch_size = 32, epochs =10,validation_split= 0.2, verbose =2,callbacks = [tensorboard])

Epoch 1/10
100/100 - 34s - loss: 0.0383 - accuracy: 0.9859 - val_loss: 0.9608 - val_accuracy: 0.5125 - 34s/epoch - 340ms/step
Epoch 2/10
100/100 - 19s - loss: 0.0015 - accuracy: 0.9997 - val_loss: 1.6085 - val_accuracy: 0.5125 - 19s/epoch - 190ms/step
Epoch 3/10
100/100 - 19s - loss: 4.2640e-04 - accuracy: 1.0000 - val_loss: 0.7699 - val_accuracy: 0.5088 - 19s/epoch - 191ms/step
Epoch 4/10
100/100 - 19s - loss: 2.7593e-04 - accuracy: 1.0000 - val_loss: 0.5993 - val_accuracy: 0.6438 - 19s/epoch - 190ms/step
Epoch 5/10
100/100 - 19s - loss: 8.0457e-04 - accuracy: 1.0000 - val_loss: 0.5647 - val_accuracy: 0.7525 - 19s/epoch - 187ms/step
Epoch 6/10
100/100 - 19s - loss: 6.4908e-04 - accuracy: 1.0000 - val_loss: 0.1079 - val_accuracy: 0.9588 - 19s/epoch - 193ms/step
Epoch 7/10
100/100 - 19s - loss: 6.9372e-05 - accuracy: 1.0000 - val_loss: 0.0102 - val_accuracy: 0.9987 - 19s/epoch - 188ms/step
Epoch 8/10
100/100 - 19s - loss: 1.5671e-04 - accuracy: 1.0000 - val_loss: 0.0012 - val_accuracy: 1.0000 - 19s/epoch - 193ms/step
Epoch 9/10
100/100 - 19s - loss: 6.2696e-05 - accuracy: 1.0000 - val_loss: 1.0221e-04 - val_accuracy: 1.0000 - 19s/epoch - 189ms/step
Epoch 10/10
100/100 - 19s - loss: 6.3716e-05 - accuracy: 1.0000 - val_loss: 8.0318e-06 - val_accuracy: 1.0000 - 19s/epoch - 193ms/step
<keras.callbacks.History at 0x7f1bd038bf50>
```

Figure20. Result of the fine-tuning trail 7

```
[ ] !tensorboard dev upload \
      --logdir logs \
      --name "Sample op-level graph" \
      --one_shot
```

```
New experiment created. View your TensorBoard at: https://tensorboard.dev/experiment/KrSnjcwWQiOrbvyDXXmi9A/

[2022-06-17T19:32:50] Started scanning logdir.
[2022-06-17T19:32:50] Total uploaded: 660 scalars, 0 tensors, 0 binary objects
[2022-06-17T19:32:50] Done scanning logdir.


Done. View your TensorBoard at https://tensorboard.dev/experiment/KrSnjcwWQiOrbvyDXXmi9A/
```

Figure21. code snippet7

Tensorboard package is imported to visualize the epoch loss, epoch accuracy , validation loss and validation accuracy.

```
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

Figure22. code snippet8

In order to visualize the prediction results on test set , a confusion matrix is used. A confusion matrix is a table that indicates how well a classification model performed on a number of test data for which the true values are known.

```
[ ]  test_path = "gdrive/My Drive/Yeni/test_batches"

[ ]  test_batches =ImageDataGenerator(preprocessing_function = tf.keras.applications.mobilenet.preprocess_input).flow_from_directory(
         directory = test_path, target_size = (224,224), batch_size =5 , shuffle = False)

     Found 50 images belonging to 2 classes.

[ ]  test_labels = test_batches.classes

[ ]  print(test_labels)

     [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1 1 1 1 1 1 1]

[ ]  predictions = new_model.predict(x = test_batches, verbose =0)
```

Figure23. code snippet9

Test set includes 25 images for each of classes, in total 50 images.

```
[59] print(predictions)

[60] test_batches.class_indices

     {'closed_eye': 0, 'open_eye': 1}

cm = confusion_matrix(y_true=test_labels ,y_pred=predictions.argmax(axis=1))

cm_plot_labels = ['0','1']
plot_confusion_matrix(cm=cm, classes = cm_plot_labels,title = 'Confusion Matrix')
```

Figure24. code snippet10

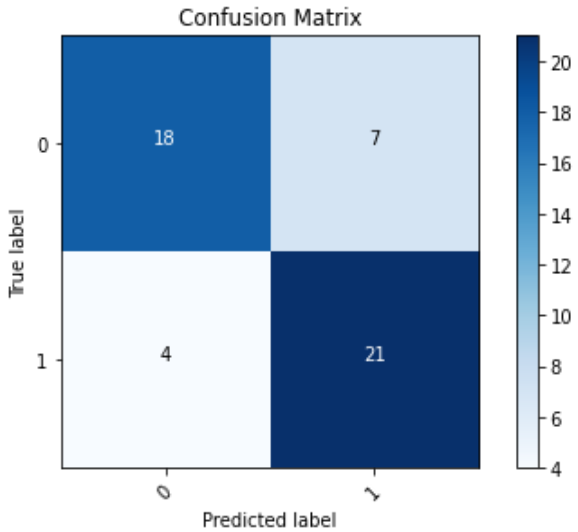


Figure25. Confusion matrix of the proposed model

Predictions are visualized with a confusion matrix as shown in the figure 13.

Figure26. code snippet10

A closed eye and an open eye image are used for prediction as shown in figure12 and figure13.

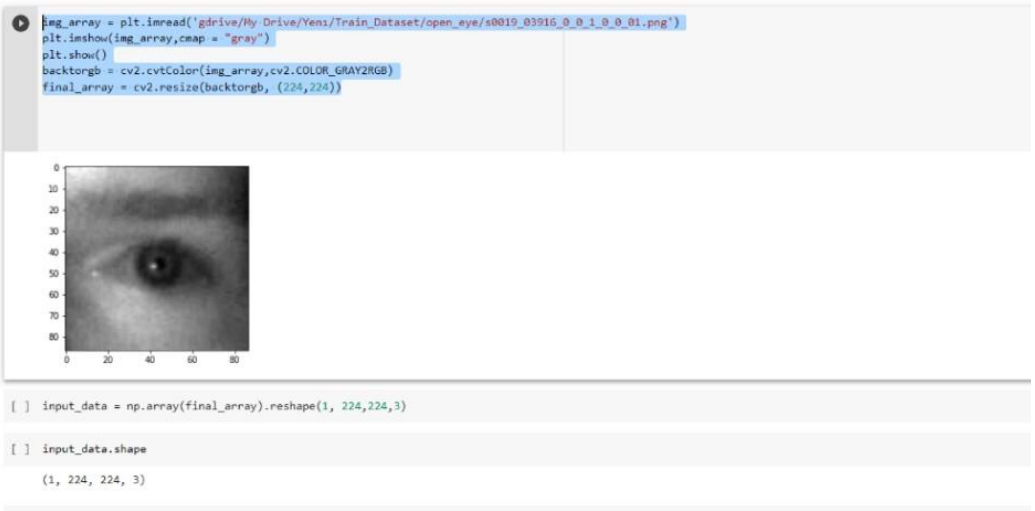


Figure27. code snippet11

Step4. Model extraction

ALARM

If the eyes are closed longer than they should be, the alarm will sound continuously to give a warning. The code below explains after determining the eye and face in the video, deciding whether it is open or closed by the model and after prediction made by model, overwriting the output video as text. Then, if the eyes have been closed longer than the specified time, the alarm will generate continuously as long as the eyes remain closed. However, in order to do this, first of all, the library named winsound must be added and the frequency and duration parameters must be defined. The rest of the code is exactly same as we did in the real-time video demo part.

In [42] :

```
import winsound
frequency = 2500
duartion = 1000
import numpy as np
import cv2
path = "haarcascade_frontalface_default.xml"
faceCascade = cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_frontalface_default.xml")

cap = cv2.VideoCapture(1)
if not cap.isOpened():
    cap = cv2.VideoCapture(0)
if not cap.isOpened():
    raise IOError("Cannot open webcam")
counter = 0
while True:
    ret,frame = cap.read()
    eye_cascade = cv2.CascaeClassifier(cv2.data.haarcascades + "haarcascade_eye_tree_eyeglasses.xml")
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    eyes = eye_cascade.detectMultiScale(gray, 1.1, 4)
    for x,y,w,h in eyes:
        roi_gray = gray[y:y+h, x:x+w]
        roi_color = frame[y:y+h, x:x+w]
        cv2.rectangle(frame, (x,y), (x+w, y+h), (0, 255, 0), 2)
        eyess = eye_cascade.detectMultiScale(roi_gray)
```

```
        if len(eyess) == 0:
            print("eyes are not detected")
        else:
            for(ex,ey,ew,eh) in eyess:
                eyes_roi = roi_color[ey: ey+eh, ex:ex +ew]

        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        print(faceCascade.empty())
        faces = faceCascade.detectMultiScale(gray, 1.1., 4)

        #Draw rectangle around the faces
        for(x, y, w, h) in faces:
            cv2.rectangle(frame, (x,y), (x+w, y+h), (0, 255, 0), 2)

        font = cv2.FONT_HERSHEY_SIMPLEX

        #Inserting the text on the video

        final_image = cv2.resize(eyes_roi, (224,224))
        final_image = np.expand_dims(final_image, axis = 0) #4th dimension is needed
        final_image = final_image/255.0

        Predictions = new_model.predict(final_image)
        if(Predictions>0):
            status = "Open Eyes"
            cv2.putText(frame, status, (150,150), font, 3, (0, 255, 0), 2, cv2.LINE_4)
            x1,y1,w1,h1 = 0, 0, 175, 175

            #To draw a black background as rectangle:
            cv2.rectangle(frame, (x1, x1), (x1 + w1, y1 + h1), (0, 0, 0), -1)
```

```
            #To draw a black background as rectangle:
            cv2.rectangle(frame, (x1, x1), (x1 + w1, y1 + h1), (0, 0, 0), -1)

            #Add the text
            cv2.putText(frame, "Active", (x1 + int(w1/10), y1 + int(h1/2)), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
        else:
            counter = counter + 1
            status = "Closed Eyes"
            cv2.putText(frame, status, (150,150), font, 3, (0, 0, 255), 2, cv2.LINE_4)
            cv2.rectangle(frame, (x,y), (x+w, y+h), (0, 0, 255), 2)
            if counter > 5:

                x1, y1, w1, h1 = 0, 0, 175, 75

                #To draw a black background as rectangle:
                cv2.rectangle(frame, (x1, x1), (x1 + w1, y1 + h1), (0, 0, 0), -1)

                #Add the text
                cv2.putText(frame, "WARNING:Sleep Alert!", (x1 + int(w1/10), y1 + int(h1/2)), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0,
                winSound.Beep(frequency, duration)
                counter = 0

        cv2.imshow("EHB 420E-Drowsiness Detection Project", frame)

        if cv2.waitKey(2) & 0xFF ==ord("q"):
            break
cap.release()
cv2.destroyAllWindows()
```

CONCLUSION

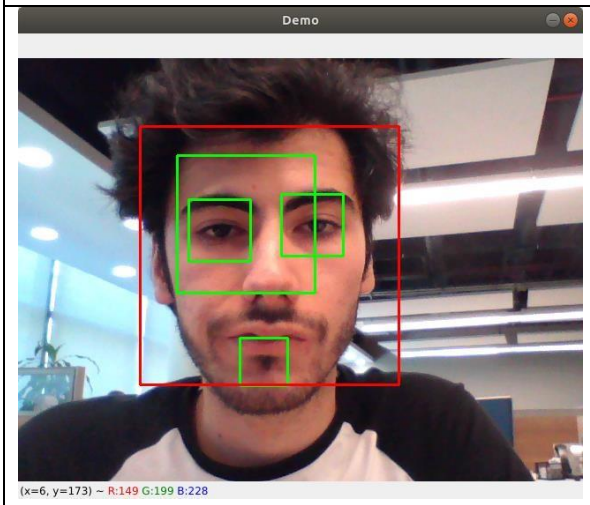
The drowsiness detection project is designed to increase safety while driving and to reduce fatal accidents. Although there are many such projects in the literature and these projects are based on making predictions based on the state of whether the eyes are closed or open and yawning. However, throughout the project, we wanted to make predictions by considering only the state of the eyes in the project. Because in many articles we researched, it is stated that people can yawn in different ways, but their eyes are always closed while sleeping. Therefore, we decided to design our project in this way in order to increase the accuracy of the project. Also, since we chose this project in a way that we can encounter in real life, we decided to do the project with real-time video. During the project, first of all, we designed the network and a model suitable for our problem is developed, then the eye and face are defined as determined by face recognition, then by successfully determining and cropping the eye from a whole photo frame, we used this as an input in the network model and, as we explained in more detail in the article, our model can decide whether the eyers are open or closed by checking the output number values –if this number is positive than, model make a prediction that the eyes are open. If not model make a prediction that the eyes are closed- Afterwards, we ensured that the alarm sound was played if the eyes were closed for sufficient time, and we successfully completed the project. We have prepared a video that shows that the code is working successfully, The photo seen below is taken from the real-time video showing the code working. However, since we could not upload this video to ninova, we wanted to share it below as a GITHUB link.

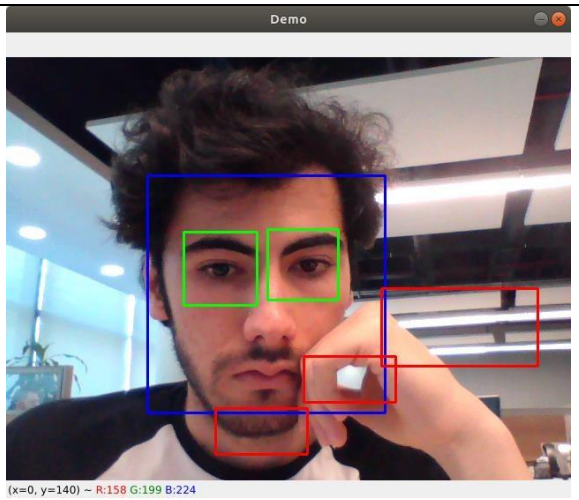
https://github.com/aerenkaradag/drowsiness_detection

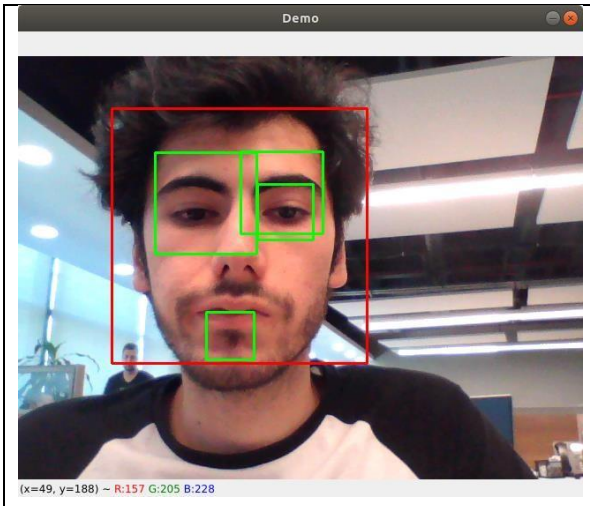
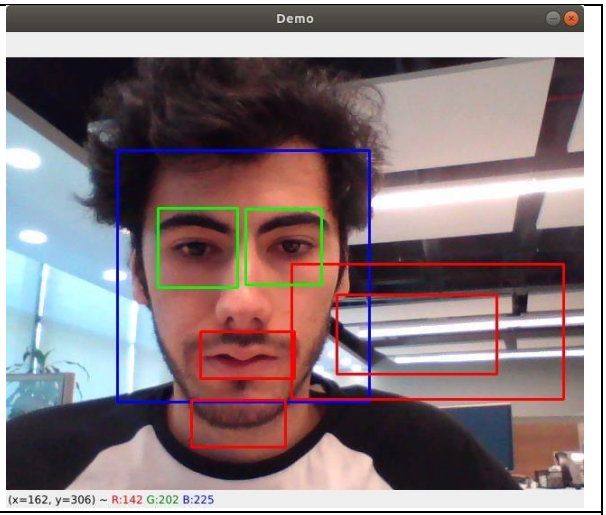
Our solutions for problems

In Fig.28, in the left pictures, there is no other suspicion that could be misclassified as an eye, although the appearance of an eye on the chin and the illusion of the real eyes can be seen. This is a common mistake among those who only use only 1 or 2 haar cascade classifiers. Adding a mouth (smile) cascade classifier to reduce fake eyes has increased the accuracy of our system by sending the right eyes frames to our model, thanks to the correct detection of the eyes, as seen in the pictures on the right. Eyes are green rectangle, face is blue rectangle, mouth and also wrong mouths are red rectangle. Wrong mouth rectangles don't matter since our model doesn't take any mouth frames.

Fig. 28





	
without smile cascade classifiers	*with smile cascade classifiers* better eye cropping

CONTRIBUTION TABLE

The Work Done	**İLKE BURÇAK**	**ÖZLEM YALÇIN**	**ALİ EREN KARADAĞ**
Selecting The Subject For The Project By Determining The Problems In Daily Life	√	√	√
Literature review	√		√
Creating The Network Model & Layers And Writing The Code		√	

Coding The Face Recognition System Using Haar Cascade And Adaboost Algorithms	√		√
Combining The Code Written For The Network And The Codes Written For Face Recognition			√
Writing The Code To Sound The Alarm	√		√
Contribute To Writing Term Paper	√	√	√