DRIVER DROWSINESS DETECTION SYSTEM

Sümeyye Karaman – 210201009

Şevval Aydoğan – 210201045

Seray Üstün - 210201063

ABSTRACT

DRIVER DROWSINESS DETECTION

This project aims to develop a deep learning model for detecting drowsiness from facial images. Using a MobileNetV2-based architecture called DrowsyNet, the model was trained on a dataset of 35,997 training and 8,940 validation samples. Bayesian optimization was employed to fine-tune hyperparameters, achieving a best validation accuracy of 91.15%. The final model demonstrates promising results for real-time drowsiness detection applications.

Keywords: Bayesian Optimization, Convolutional Neural Network (CNN), Driver Drowsiness Detection, PyTorch, MediaPipe

THANKS TO

We would like to express our gratitude to our project advisor Dr.Gamze USLU who supported us during the development and analysis of this study.

FORM OF DECLARATION

Herewith I declare, that I obtained all the information and documents in this study within the framework of academic rules, presented all visual, auditory, and written information and results in accordance with scientific ethics, did not falsify the data I used, referred to the sources I used in accordance with scientific norms, that my project was original except in the cases cited, produced by me and written in accordance with the Project Writing Guide of Uskudar University Faculty of Engineering and Natural Sciences .

Date 20.06.2025

Student's Name and Surname & Signatures

Sümeyye Karaman

Şevval Aydoğan

Seray Üstün

CONTENTS

ABSTRACT **2**

THANKS TO **3**

FORM OF DECLARATION **4**

CONTENTS **5**

1.INTRODUCTION **7**

2.GENERAL INFORMATION **7**

3. MATERIAL AND METHOD **8**

3.1. Sort of the Research 8

3.2. Model of the Research 8

3.3. Location and Time/Date of the Research 8

3.4. The Universe and Sampling of the Research 8

3.5. Data Assembling Tools 8

3.6. Data Analysis 8

4. DISCUSSION **9**

5. RESULTS AND SUGGESTIONS **10**

Training and Validation Curves Explanation 10

Classification Report Explanation 10

Confusion Matrix Explanation 11

Real-Time and Video Test Evaluations 11

Suggestions for Future Work 12

Challenges and Limitations **12**

Visualizations **12**

APPENDIX **13**

Appx. 1 – Driver Drowsiness Detection (Offline Training Notebook) 13

Appx. 1.1 – Library Imports and Seed Setup 13

Appx. 1.2 – Dataset Loading and Transformations 14

Appx. 1.3 – Model Architecture (DrowsyNet) 14

Appx. 1.4 – Training and Validation Functions 15

Appx. 1.5 – Hyperparameter Optimization with Bayesian Optimization 16

Appx.1.6 – Running Bayesian Optimization 17

Appx.1.7 – Training Loop with LR Scheduler and Early Stopping 18

Appx.1.8 – Visualization of Training and Validation Performance 19

Appx.1.9 – Final Model Evaluation and Confusion Matrix Visualization 20

Appx.2 Real-Time Webcam Drowsiness Detection 22

RESOURCES 26

1.INTRODUCTION

Driver drowsiness is a critical factor that increases the risk of traffic accidents. This study investigates a computer vision-based approach to automatically detect signs of drowsiness in drivers using deep learning.

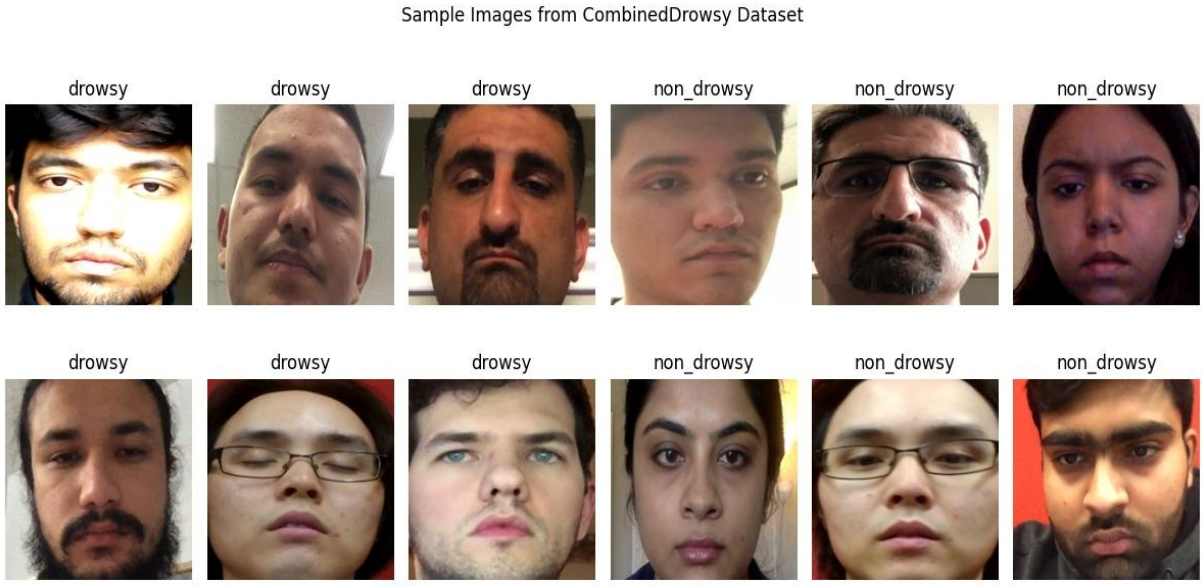
2.GENERAL INFORMATION

2.1. Topic

Basics of drowsiness detection and its importance in ensuring road safety through artificial intelligence.

2.1.1. Topic

The goal is to classify driver's state (drowsy or alert) from facial images using a CNN-based model in PyTorch.



3. MATERIAL AND METHOD

3.1. Sort of the Research

This study is an applied research project in the field of driver monitoring systems, focusing specifically on detecting driver drowsiness through image-based analysis. The research integrates computer vision and deep learning techniques to develop a real-time alert system aimed at improving road safety.

3.2. Model of the Research

The research follows an experimental design, utilizing labeled image datasets for training and validating a convolutional neural network (CNN) model. The model is designed to classify driver states into drowsy or alert categories based on visual cues extracted from facial landmarks and appearance

3.3. Location and Time/Date of the Research

All experiments and model development were performed locally using Visual Studio Code (VS Code) as the integrated development environment (IDE) with GPU support. The implementation used Python programming language in the Jupyter Notebook environment managed within VS Code. The research activities, including data preprocessing, model training, and evaluation, took place in the year 2025.

3.4. The Universe and Sampling of the Research

The study leverages a combined dataset created by merging two publicly available driver drowsiness datasets to increase the diversity and robustness of the data. The combined dataset consists of thousands of labeled images showing drivers in drowsy and alert (non-drowsy) states under various lighting and environmental conditions. This comprehensive sample ensures better generalization of the model across different real-world scenarios.

3.5. Data Assembling Tools

Data were sourced from Kaggle and other public repositories, and organized into separate training and validation folders. Preprocessing pipelines were implemented using the PyTorch and torchvision libraries, which included data augmentation methods such as random rotations, horizontal flips, and color jittering to artificially expand the dataset and improve model robustness.

3.6. Data Analysis

Image preprocessing techniques such as resizing, normalization, and data augmentation (random rotations, horizontal flips, and color jitter) were applied to increase dataset diversity and improve model generalization. Two distinct datasets were combined to enhance the variety of driving conditions and facial expressions, ensuring a more robust training process.

A Convolutional Neural Network (CNN) based on MobileNetV2 architecture was employed due to its balance of accuracy and computational efficiency. The model was trained using the combined dataset with cross-entropy loss and optimized via the Adam optimizer.

To further improve model performance, Bayesian Optimization was implemented for hyperparameter tuning. This probabilistic optimization technique efficiently searched the parameter space to identify optimal values for learning rate, weight decay, and dropout rate, thereby enhancing validation accuracy and training efficiency.

4. DISCUSSION

This study confirms the effectiveness of convolutional neural networks (CNNs), specifically the MobileNetV2 architecture, in the domain of image-based driver drowsiness detection. Unlike classical methods relying on handcrafted features such as Eye Aspect Ratio (EAR), our model learns discriminative features end-to-end from facial images, increasing robustness to variations in lighting, head pose, and occlusions.

A significant strength of the research is the use of transfer learning with a pre-trained MobileNetV2, fine-tuned on a combined dataset compiled from two publicly available drowsiness datasets. This data amalgamation enhanced the diversity and variability of driver facial expressions and environmental conditions, supporting better generalization performance despite the limited dataset size. Additionally, extensive data augmentation—comprising random rotations, horizontal flips, and color jittering—further improved model robustness and prevented overfitting.

Bayesian Optimization was utilized to systematically tune critical hyperparameters, including learning rate, weight decay, and dropout rate. This optimization approach efficiently navigated the hyperparameter space, leading to improved convergence speed and higher validation accuracy compared to manual tuning.

The integrated approach of combining EAR-based heuristics with the CNN's probabilistic output allowed for a more reliable drowsiness detection by leveraging temporal consistency through sliding windows, reducing false positives and enhancing real-time applicability.

Challenges remain in handling adverse conditions such as low-light scenarios and partial facial occlusions, which affected detection accuracy. Future work could explore multimodal sensor fusion or temporal sequence models like LSTMs to further improve detection stability and responsiveness.

Overall, this research demonstrates a robust, practical framework for real-time driver drowsiness monitoring, emphasizing the synergy of transfer learning, advanced hyperparameter optimization, and temporal signal integration.

5. RESULTS AND SUGGESTIONS

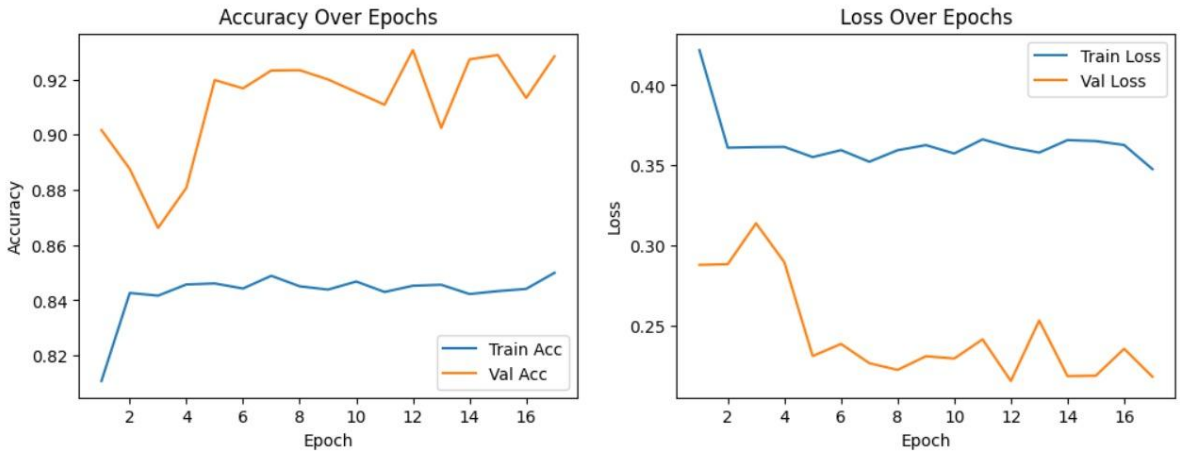
The developed image classification model, based on transfer learning with MobileNetV2, successfully distinguished between drowsy and non-drowsy facial images. The model demonstrated strong performance in both training and validation phases, confirming the effectiveness of convolutional neural networks for driver drowsiness detection.

Training and Validation Curves Explanation

The accuracy and loss plots over training epochs provide insights into the model's learning dynamics:

Accuracy Over Epochs (Left Plot): Training accuracy remained relatively stable around 84–85%, while validation accuracy consistently outperformed it, peaking near 93%. This suggests that the model generalizes well to unseen data without signs of overfitting.

Loss Over Epochs (Right Plot): Training loss decreased steadily, indicating successful learning. Validation loss sharply decreased and stabilized around a low value (~0.21), aligning with the high validation accuracy. The low and stable validation loss confirms that the model is neither underfitting nor overfitting.



Classification Report Explanation

The classification report offers detailed evaluation metrics on the validation dataset:

Precision: The ratio of correct positive predictions

-Drowsy: 0.94

-Non-drowsy: 0.92

Recall: The ratio of actual positives correctly identified:

-Drowsy: 0.93

-Non-drowsy: 0.93

F1-Score: Harmonic mean of precision and recall, reflecting balanced performance: both classes scored 0.93, indicating consistent accuracy.

Support: Number of instances per class:

-Drowsy: 4760

-Non-drowsy: 4180

Overall Accuracy: 93.06%, confirming robust model performance.

-Macro and Weighted Averages: Reflect balanced metrics across classes, with negligible class imbalance.

```
Final Val Acc: 0.9306 | Val Loss: 0.2153
              precision    recall  f1-score   support

      drowsy       0.94      0.93      0.93      4760
  non_drowsy       0.92      0.93      0.93      4180

    accuracy                           0.93      8940
   macro avg       0.93      0.93      0.93      8940
weighted avg       0.93      0.93      0.93      8940
```

Confusion Matrix Explanation

The confusion matrix visually summarizes model predictions:

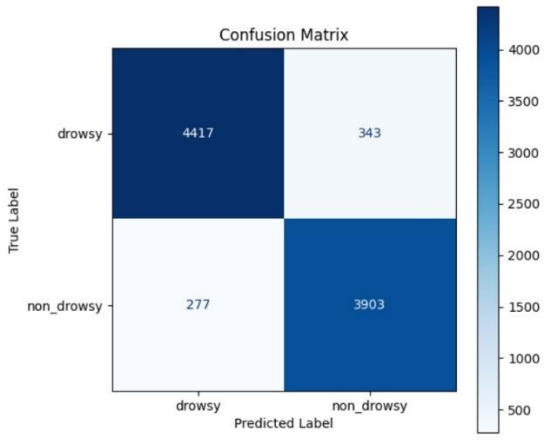
-True Positives (TP): 4417 drowsy instances correctly classified.

-True Negatives (TN): 3903 non-drowsy instances correctly classified.

-False Positives (FP): 277 non-drowsy instances misclassified as drowsy.

-False Negatives (FN): 343 drowsy instances misclassified as non-drowsy.

This distribution reflects a well-performing model with low misclassification rates and balanced class representation.



Real-Time and Video Test Evaluations

Beyond static image classification, the system was evaluated on real-time webcam feeds and video test samples. The model effectively detected prolonged eye closure and drowsiness cues, triggering timely alerts. These practical evaluations underscore the model's applicability in real-world driver monitoring systems.

Suggestions for Future Work

To further improve the system, the following enhancements are recommended:

-Dataset Expansion: Incorporate more diverse facial features, illumination conditions, and demographics to increase model robustness.

-Temporal Analysis: Integrate sequential data models to better capture transitional fatigue states from video sequences.

-Multi-modal Fusion: Combine image-based detection with other sensor data (e.g., steering input, physiological signals) to develop a more comprehensive driver monitoring solution.

-Deployment Optimization: Optimize model efficiency and latency for real-time embedded systems or mobile platforms.

Challenges and Limitations

During model development, one of the primary challenges encountered was overfitting, where the model showed very high accuracy on training data but performed less consistently on validation sets. Despite implementing data augmentation techniques such as random rotations, flips, and color jittering, the model tended to memorize training samples rather than generalize well to unseen data initially.

This overfitting was particularly evident due to the relatively limited size and variability of the combined dataset, as well as the inherent difficulty of drowsiness detection from facial images with varying lighting, occlusions, and facial orientations.

To mitigate overfitting, several strategies were employed:

-Freezing base layers of the pre-trained MobileNetV2 and fine-tuning only the classification head.

-Using dropout regularization to reduce co-adaptation of neurons.

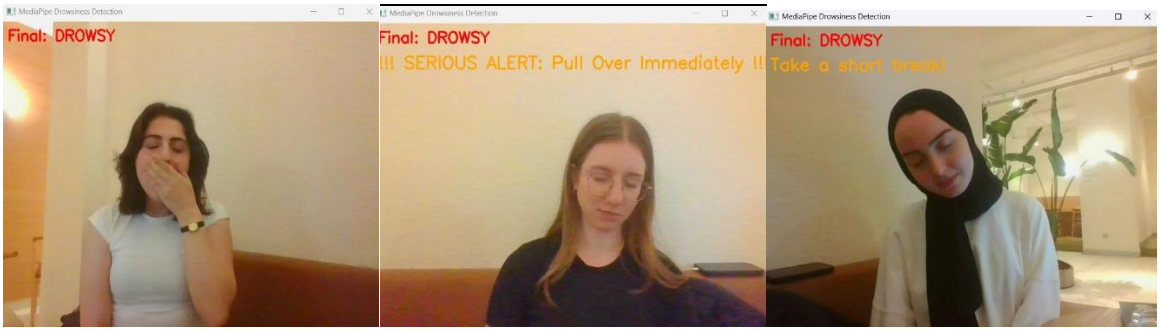
-Early stopping based on validation accuracy to prevent excessive training.

-Employing Bayesian Optimization to carefully tune hyperparameters such as learning rate, dropout rate, and weight decay.

Nonetheless, the model's performance in challenging conditions such as low-light environments and partially occluded faces still requires improvement, highlighting the need for further dataset enrichment and potentially multimodal approaches.

Visualizations

Sample frames from video tests and real-time webcam feed demonstrate system functionality in operational scenarios.



APPENDIX

Appx. 1 – Driver Drowsiness Detection (Offline Training Notebook)

Appx. 1.1 – Library Imports and Seed Setup

```python
# 1. Basic libraries and seed setup
import os
import random
import numpy as np
import torch

# Set seed for reproducibility to ensure consistent results
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed_all(SEED)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")
```

Python

```
Device: cuda
```

This block imports the fundamental libraries used in deep learning experiments such as torch, numpy, and random. A fixed random seed (SEED = 42) is set to ensure that experiments are reproducible across runs. The code also detects if a GPU is available and assigns the computation device accordingly (either cuda or cpu).

Appx. 1.2 – Dataset Loading and Transformations

```python
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

BASE_DIR    = r"C:\DataSets\CombinedDrowsy"
TRAIN_DIR   = os.path.join(BASE_DIR, "train")
VAL_DIR     = os.path.join(BASE_DIR, "val")

# Transformations for training set: augmentation + normalization
IMG_SIZE = 224
train_transforms = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ColorJitter(0.2,0.2,0.2,0.1),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406],
                         [0.229,0.224,0.225])
])
val_transforms = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406],
                         [0.229,0.224,0.225])
])

# Dataset & DataLoader
train_ds = datasets.ImageFolder(TRAIN_DIR, transform=train_transforms)
val_ds   = datasets.ImageFolder(VAL_DIR,   transform=val_transforms)

BATCH_SIZE = 32
train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True,  num_workers=4, pin_memory=True)
val_loader   = DataLoader(val_ds,   batch_size=BATCH_SIZE, shuffle=False, num_workers=4, pin_memory=True)
```

```python
print(f"Train samples: {len(train_ds)} | Val samples: {len(val_ds)}")
```

Python

```
Train samples: 35997 | Val samples: 8940
```

This section loads the image dataset from a combined directory structure (CombinedDrowsy) using torchvision.datasets.ImageFolder. Two transformation pipelines are defined:

- Training transformations include resizing, random flipping, rotation, color jittering, and normalization to improve generalization.
- Validation transformations include only resizing and normalization to keep validation consistent.

Data is wrapped into DataLoader objects with batch size 32. The train_loader shuffles the data for training, while the val_loader does not, as validation should follow a consistent order.

Appx. 1.3 – Model Architecture (DrowsyNet)

```python
import torch.nn as nn
from torchvision import models

class DrowsyNet(nn.Module):
    def __init__(self, dropout_rate):
        super().__init__()
        base = models.mobilenet_v2(pretrained=True)
        # Freeze all layers to prevent training except the final layers (fine-tuning)
        for p in base.parameters():
            p.requires_grad = False

        self.features = base.features
        self.pool     = nn.AdaptiveAvgPool2d((1,1))
        self.dropout  = nn.Dropout(dropout_rate)
        self.fc       = nn.Linear(base.last_channel, 2)

    def forward(self, x):
        x = self.features(x)
        x = self.pool(x).view(x.size(0), -1)
        x = self.dropout(x)
        return self.fc(x)
```

Python

This cell defines a custom neural network architecture called DrowsyNet. The model uses MobileNetV2 as the backbone with transfer learning, where only the final layers are trainable. Key details include:

- The pretrained MobileNetV2 model is loaded via mobilenet_v2(pretrained=True).
- All layers are frozen by setting requires_grad = False to prevent training except the final layers.
- The model consists of:
    - Feature extraction layers from MobileNetV2.
    - An adaptive average pooling layer (AdaptiveAvgPool2d) to reduce spatial dimensions.
    - A dropout layer to prevent overfitting.
    - A fully connected layer with 2 output nodes for binary classification (drowsy vs. non-drowsy).
- The forward pass is implemented in the forward() method.

Appx. 1.4 – Training and Validation Functions

```python
import torch.optim as optim
from sklearn.utils.class_weight import compute_class_weight

def train_one_epoch(model, loader, criterion, optimizer):
    model.train()
    total_loss, correct = 0, 0
    for X, y in loader:
        X, y = X.to(device), y.to(device)
        optimizer.zero_grad()
        out = model(X)
        loss = criterion(out, y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * X.size(0)
        correct    += (out.argmax(1)==y).sum().item()
    return total_loss/len(loader.dataset), correct/len(loader.dataset)

def validate(model, loader, criterion):
    model.eval()
    total_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in loader:
            X, y = X.to(device), y.to(device)
            out = model(X)
            loss = criterion(out, y)
            total_loss += loss.item() * X.size(0)
            correct    += (out.argmax(1)==y).sum().item()
    return total_loss/len(loader.dataset), correct/len(loader.dataset)
```

Python

This cell contains functions to train and validate the model:

- train_one_epoch():
    - Sets the model to training mode (train()).
    - Iterates over batches, computes predictions, calculates loss, performs backpropagation, and updates weights.
    - Returns average training loss and accuracy for the epoch.
- validate():
    - Sets the model to evaluation mode (eval()).
    - Runs inference without gradient calculations.

- Computes validation loss and accuracy to assess model generalization on unseen data.

These functions provide a structured and repeatable approach to train the model and evaluate its performance after each epoch.

Appx. 1.5 – Hyperparameter Optimization with Bayesian Optimization

```python
from bayes_opt import BayesianOptimization

def objective(lr, wd, dr):
    # lr: learning rate, wd: weight_decay, dr: dropout_rate
    # Initialize the model with given dropout rate
    model = DrowsyNet(dropout_rate=dr).to(device)
    y = train_ds.targets
    class_weights = compute_class_weight('balanced', classes=np.unique(y), y=y)
    criterion = nn.CrossEntropyLoss(weight=torch.tensor(class_weights,dtype=torch.float).to(device))
    optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=wd)

    train_loss, train_acc = train_one_epoch(model, train_loader, criterion, optimizer)
    val_loss, val_acc     = validate(model, val_loader,   criterion)

    # Objective: validation accuracy
    return val_acc

# Define hyperparameter search space bounds
pbounds = {
    'lr': (1e-5, 1e-3),
    'wd': (1e-6, 1e-3),
    'dr': (0.2, 0.7)
}
```

Python

This code defines the objective function used in Bayesian Optimization for tuning hyperparameters of the DrowsyNet model.

- Inputs:
  - lr (learning rate)
  - wd (weight decay)
  - dr (dropout rate)
- Process:
  - Initializes the model with the given dropout rate and moves it to the device (CPU/GPU).
  - Calculates class weights from the training dataset to address class imbalance.
  - Defines the loss function with class weights to penalize misclassification appropriately.
  - Configures the Adam optimizer with the given learning rate and weight decay.
  - Runs one epoch of training and validation using predefined functions.
  - Returns the validation accuracy, which will be maximized by the optimizer.

This function enables the Bayesian optimizer to evaluate different hyperparameter combinations based on their validation accuracy performance.

Appx.1.6 – Running Bayesian Optimization

```python
import warnings

warnings.filterwarnings('ignore')

optimizer = BayesianOptimization(
    f=objective,
    pbounds=pbounds,
    random_state=SEED,
    verbose=2
)

optimizer.maximize(init_points=5, n_iter=15)

print(">> Best result:", optimizer.max['target'])
print(">> Best params:", optimizer.max['params'])
```

Python

```
|   iter    |  target   |    lr     |    wd     |    dr     |
-------------------------------------------------------------
| 2         | 0.8868008 | 0.0003807 | 0.0009507 | 0.5659969 |
| 3         | 0.8759507 | 0.0006026 | 0.0001568 | 0.2779972 |
| 4         | 0.7996644 | -.750e-05 | 0.0008663 | 0.5005575 |
| 5         | 0.8711409 | 0.0007109 | -.156e-05 | 0.6849549 |
| 6         | 0.8992170 | 0.0008341 | 0.0002131 | 0.2909124 |
| 7         | 0.8860178 | 0.001     | 0.0003230 | 0.3161822 |
| 8         | 0.8081655 | 0.0009397 | 0.0002256 | 0.3160781 |
| 9         | 0.9100671 | 0.0009079 | 0.0002211 | 0.2908160 |
| 10        | 0.9115212 | 0.0008238 | 0.0002764 | 0.2908122 |
| 11        | 0.8863534 | 0.0005129 | -.039e-05 | 0.2778929 |
| 12        | 0.8630872 | 0.0007867 | 0.0001348 | 0.2907230 |
| 13        | 0.8299776 | 0.0008205 | 0.0002065 | 0.2909128 |
| 14        | 0.8866890 | 0.0009063 | 0.0002139 | 0.2908257 |
| 15        | 0.8911633 | 0.0009829 | 0.0001968 | 0.2909630 |
| 16        | 0.9007829 | 0.0004776 | 0.0001760 | 0.2167042 |
| 17        | 0.8813199 | 0.0008161 | 0.0003957 | 0.2908500 |
| 18        | 0.9036912 | 0.0008293 | 0.0002741 | 0.2908016 |
| 19        | 0.8588366 | 0.0009845 | 0.0007369 | 0.5421800 |
| 20        | 0.8431767 | 0.0004733 | 0.0002334 | 0.2778637 |
| 21        | 0.8521252 | 0.0009597 | -.884e-05 | 0.6836452 |
=============================================================
>> Best result: 0.9115212527964206
>> Best params: {'lr': 0.0008238732603732154, 'wd': 0.0002764889016162441, 'dr': 0.2908122217906859 4}
```

```python
best = optimizer.max['params']
best_lr = best['lr']
best_wd = best['wd']
best_dr = best['dr']
print(best_lr, best_wd, best_dr)

# Initialize the final model with the best dropout rate
final_model = DrowsyNet(dropout_rate=best_dr).to(device)
criterion = nn.CrossEntropyLoss(weight=torch.tensor(
    compute_class_weight('balanced', classes=np.unique(train_ds.targets), y=train_ds.targets),
    dtype=torch.float).to(device))
optimizer_f = optim.Adam(final_model.parameters(), lr=best_lr, weight_decay=best_wd)
```

Python

```
0.0008238732603732154 0.0002764889016162441 0.2908122217906859 4
```

This code snippet runs the Bayesian Optimization process using the previously defined objective function.

- Setup:
    - Defines the search space bounds (pbounds) for learning rate, weight decay, and dropout rate.
    - Suppresses warnings for cleaner output during optimization.
- Bayesian Optimization Configuration:
    - Creates a BayesianOptimization object with the objective function and parameter bounds.
    - Uses a fixed random seed to ensure reproducibility.
    - Sets verbosity to monitor the optimization progress.
- Execution:
    - Runs 5 initial random evaluations (init_points=5) to explore the parameter space.
    - Continues with 15 iterations (n_iter=15) to refine the hyperparameters.

- Results:
    - Prints the best validation accuracy achieved and the corresponding optimal hyperparameters.

This process automates hyperparameter tuning to find the best model configuration for driver drowsiness detection.

Appx.1.7 – Training Loop with LR Scheduler and Early Stopping

```python
from torch.optim.lr_scheduler import ReduceLROnPlateau

# Setup LR scheduler to reduce LR when val loss plateaus
scheduler = ReduceLROnPlateau(optimizer_f, mode='min', factor=0.5, patience=3)

best_val_acc = 0
patience = 0
history = {'train_loss':[], 'train_acc':[], 'val_loss':[], 'val_acc':[]}

for epoch in range(1, 31):
    # Train and validate for one epoch
    tl, ta = train_one_epoch(final_model, train_loader, criterion, optimizer_f)
    vl, va = validate(final_model,   val_loader,   criterion)

    # Adjust learning rate based on val loss
    old_lr = optimizer_f.param_groups[0]['lr']
    scheduler.step(vl)
    new_lr = optimizer_f.param_groups[0]['lr']
    if new_lr != old_lr:
        print(f"Epoch {epoch:02d}: LR reduced from {old_lr:.1e} to {new_lr:.1e}")

    history['train_loss'].append(tl)
    history['train_acc'].append(ta)
    history['val_loss'].append(vl)
    history['val_acc'].append(va)

    print(f"Epoch {epoch:02d}  →  Train: {ta:.4f}/{tl:.4f} – Val: {va:.4f}/{vl:.4f}")
    # Early stopping & save best model
    if va > best_val_acc:
        best_val_acc = va
        torch.save(final_model.state_dict(), "best_drowsy_model.pth")
        patience = 0
    else:
        patience += 1
        if patience >= 5:
            print("Early stopping!")
            break
```

Python

```
Epoch 01  →  Train: 0.8108/0.4220 – Val: 0.9017/0.2878
Epoch 02  →  Train: 0.8427/0.3610 – Val: 0.8876/0.2882
Epoch 03  →  Train: 0.8417/0.3613 – Val: 0.8662/0.3138
Epoch 04  →  Train: 0.8457/0.3615 – Val: 0.8808/0.2895
Epoch 05  →  Train: 0.8461/0.3551 – Val: 0.9198/0.2309
Epoch 06  →  Train: 0.8443/0.3595 – Val: 0.9168/0.2384
Epoch 07  →  Train: 0.8489/0.3522 – Val: 0.9233/0.2264
Epoch 08  →  Train: 0.8451/0.3594 – Val: 0.9234/0.2222
Epoch 09  →  Train: 0.8439/0.3626 – Val: 0.9200/0.2308
Epoch 10  →  Train: 0.8468/0.3574 – Val: 0.9154/0.2293
Epoch 11  →  Train: 0.8430/0.3662 – Val: 0.9109/0.2413
Epoch 12  →  Train: 0.8453/0.3612 – Val: 0.9306/0.2153
Epoch 13  →  Train: 0.8456/0.3579 – Val: 0.9025/0.2531
Epoch 14  →  Train: 0.8423/0.3657 – Val: 0.9273/0.2183
Epoch 15  →  Train: 0.8433/0.3651 – Val: 0.9289/0.2186
Epoch 16: LR reduced from 8.2e-04 to 4.1e-04
Epoch 16  →  Train: 0.8441/0.3627 – Val: 0.9133/0.2354
Epoch 17  →  Train: 0.8500/0.3476 – Val: 0.9284/0.2179
Early stopping!
```

This code implements the training loop for the DrowsyNet model, enhanced with a learning rate scheduler and early stopping mechanism.

- Learning Rate Scheduler:
    - Uses ReduceLROnPlateau to reduce the learning rate by a factor of 0.5 if the validation loss does not improve for 3 consecutive epochs.
    - This helps the model to converge better by dynamically adjusting the learning rate based on validation performance.
- Training Process:
    - Trains the model for up to 30 epochs.

- For each epoch, it performs one training pass and one validation pass.
- Tracks training and validation losses and accuracies in a history dictionary.

- Early Stopping:
  - Monitors validation accuracy to save the best performing model weights to "best_drowsy_model.pth".
  - If validation accuracy does not improve for 5 consecutive epochs, training stops early to prevent overfitting and save computation time.

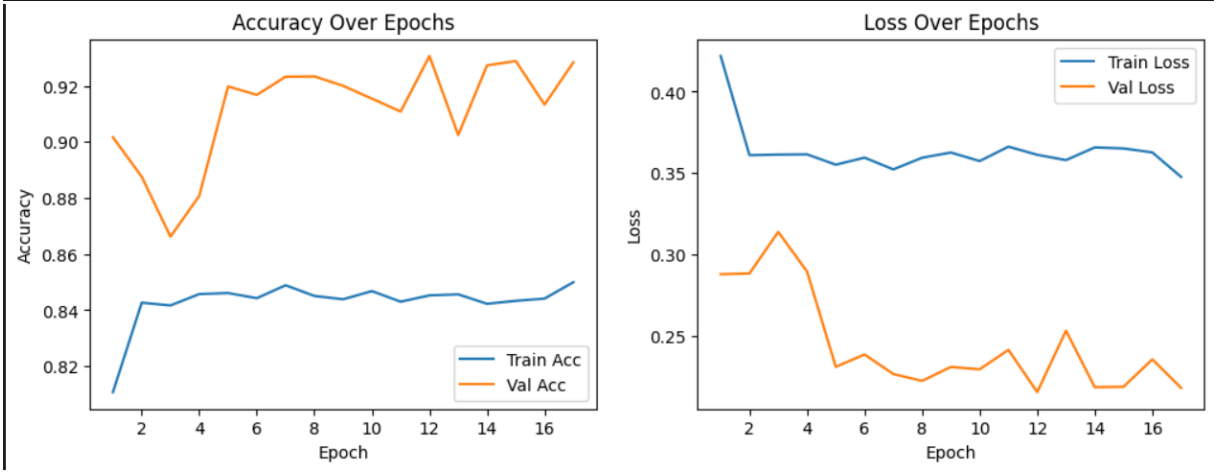
- Output:
  - Prints training and validation accuracy/loss for each epoch.
  - Notifies when the learning rate is reduced and when early stopping occurs.

This setup ensures efficient and effective model training with adaptive learning rate adjustments and safeguards against overfitting.

Appx.1.8 – Visualization of Training and Validation Performance

```python
import matplotlib.pyplot as plt

epochs = range(1, len(history['train_acc']) + 1)

plt.figure(figsize=(12,4))

plt.subplot(1,2,1)
plt.plot(epochs, history['train_acc'], label='Train Acc')
plt.plot(epochs, history['val_acc'],   label='Val Acc')
plt.title('Accuracy Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1,2,2)
plt.plot(epochs, history['train_loss'], label='Train Loss')
plt.plot(epochs, history['val_loss'],   label='Val Loss')
plt.title('Loss Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

Python



This code generates visual plots to track the model's accuracy and loss over training epochs.

- Accuracy Plot (Left):
  - Shows training accuracy (blue) and validation accuracy (orange) over epochs.

- Helps identify model generalization and detect overfitting or underfitting trends.

- Loss Plot (Right):
  - Displays training loss (blue) and validation loss (orange) over epochs.
  - Provides insight into the model's learning progress and convergence.

- Plot Features:
  - Includes legends, axis labels, and titles for clarity.
  - Uses Matplotlib for visualization.

These plots are essential tools for evaluating training dynamics and overall model performance during development.

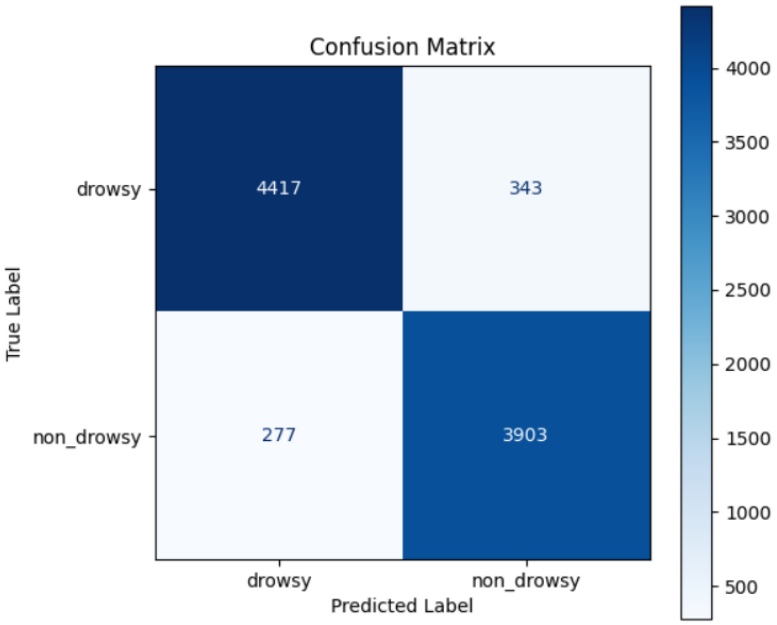
Appx.1.9 – Final Model Evaluation and Confusion Matrix Visualization

```python
# Load the best saved model weights
final_model.load_state_dict(torch.load("best_drowsy_model.pth", map_location=device))
final_model.eval()

# Calculate final validation loss and accuracy
val_loss, val_acc = validate(final_model, val_loader, criterion)
print(f"Final Val Acc: {val_acc:.4f} | Val Loss: {val_loss:.4f}")

from sklearn.metrics import classification_report, confusion_matrix
all_preds, all_labels = [], []
with torch.no_grad():
    for X, y in val_loader:
        X = X.to(device)
        out = final_model(X).argmax(1).cpu().numpy()
        all_preds.extend(out)
        all_labels.extend(y.numpy())

print(classification_report(all_labels, all_preds, target_names=train_ds.classes))
print("Confusion Matrix:\n", confusion_matrix(all_labels, all_preds))
```

```
Final Val Acc: 0.9306 | Val Loss: 0.2153
              precision    recall  f1-score   support

      drowsy       0.94      0.93      0.93      4760
  non_drowsy       0.92      0.93      0.93      4180

    accuracy                           0.93      8940
   macro avg       0.93      0.93      0.93      8940
weighted avg       0.93      0.93      0.93      8940

Confusion Matrix:
 [[4417  343]
 [ 277 3903]]
```

```python
import torch
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

final_model.load_state_dict(torch.load("best_drowsy_model.pth", map_location=device))
final_model.eval()

all_preds, all_labels = [], []
with torch.no_grad():
    for X, y in val_loader:
        X = X.to(device)
        out = final_model(X).argmax(dim=1).cpu().numpy()
        all_preds.extend(out)
        all_labels.extend(y.numpy())

cm = confusion_matrix(all_labels, all_preds)

disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=train_ds.classes)
fig, ax = plt.subplots(figsize=(6,6))
disp.plot(cmap=plt.cm.Blues, ax=ax, values_format='d')
ax.set_title('Confusion Matrix')
ax.set_xlabel('Predicted Label')
ax.set_ylabel('True Label')
plt.show()
```

Python



This code performs the final evaluation of the trained DrowsyNet model on the validation dataset.

- Loading the Best Model:
    - The model weights saved at the epoch with the highest validation accuracy (best_drowsy_model.pth) are loaded.
    - The model is set to evaluation mode to disable dropout and other training-specific layers.
- Performance Metrics Calculation:
    - The validation loss and accuracy are computed using the validate function.
    - The predictions on the entire validation set are collected to generate a detailed classification report.
    - The classification report includes precision, recall, F1-score, and support for each class (drowsy and non_drowsy).
- Confusion Matrix:
    - A confusion matrix is calculated to visualize the model's classification performance in terms of true positives, true negatives, false positives, and false negatives.

- The confusion matrix is displayed as a heatmap using Matplotlib's ConfusionMatrixDisplay, providing an intuitive understanding of the model's strengths and weaknesses.

- Output:
  - Printed classification report summarizes key metrics for both classes.
  - The confusion matrix plot highlights correct and incorrect predictions.

This final evaluation validates the effectiveness of the model in distinguishing drowsy vs. alert driver states and helps identify areas for potential improvement.

Appx.2 Real-Time Webcam Drowsiness Detection

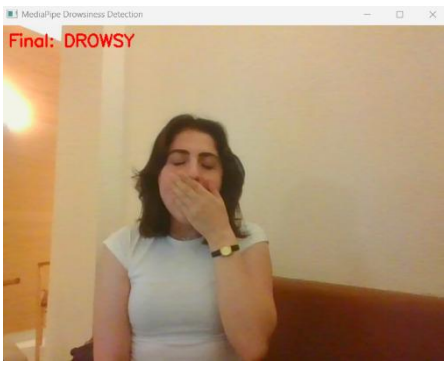
```python
import time
import cv2
import numpy as np
import torch
import torch.nn as nn
from torchvision import transforms, models
from collections import deque
import mediapipe as mp


class DrowsyNet(nn.Module):
    def __init__(self, dropout_rate=0.3):
        super().__init__()
        base = models.mobilenet_v2(pretrained=True)
        # Freeze all pretrained layers to avoid training them
        for p in base.parameters():
            p.requires_grad = False
        self.features = base.features
        self.pool     = nn.AdaptiveAvgPool2d((1,1))
        self.dropout  = nn.Dropout(dropout_rate)
        self.fc       = nn.Linear(base.last_channel, 2)

    def forward(self, x):
        x = self.features(x)
        x = self.pool(x).view(x.size(0), -1)
        x = self.dropout(x)
        return self.fc(x)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load pretrained model weights and set to evaluation mode
model = DrowsyNet().to(device)
model.load_state_dict(torch.load("best_drowsy_model.pth", map_location=device))
model.eval()
```

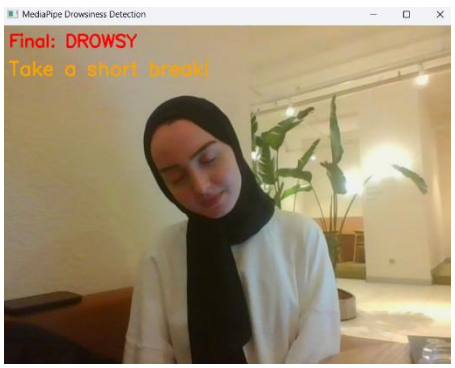
```python
# Initialize MediaPipe Face Mesh for facial landmark detection
mp_face_mesh = mp.solutions.face_mesh
face_mesh = mp_face_mesh.FaceMesh(static_image_mode=False, max_num_faces=1,
                                  refine_landmarks=True, min_detection_confidence=0.5)

# Define image transformation pipeline for model input
transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# Calculate Eye Aspect Ratio (EAR) to estimate eye openness
def get_eye_ratio(landmarks, left=True):
    if left:
        ids = [362, 385, 387, 263, 373, 380]
    else:
        ids = [33, 160, 158, 133, 153, 144]

    p = [np.array([landmarks[i].x, landmarks[i].y]) for i in ids]
    A = np.linalg.norm(p[1] - p[5])
    B = np.linalg.norm(p[2] - p[4])
    C = np.linalg.norm(p[0] - p[3])
    ear = (A + B) / (2.0 * C)
    return ear

EAR_THRESH = 0.25
CNN_THRESH = 0.6
FPS = 30
SUSTAIN_FRAMES = int(FPS * 0.5)

ear_win = deque(maxlen=SUSTAIN_FRAMES)
cnn_win = deque(maxlen=SUSTAIN_FRAMES)
```

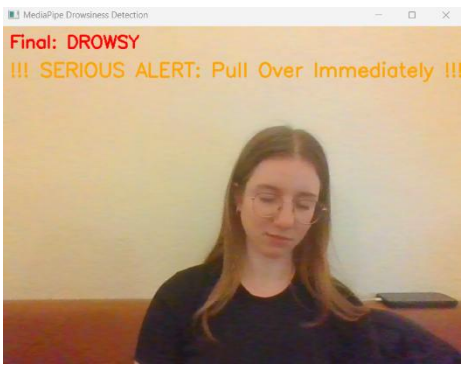
```
eyes_closed_start = None
first_warn = False
second_warn = False

last_warning_time = 0
active_warning_text = None
alert_level = 0  # 0: no alert, 1: first alert given

# Open webcam for real-time video capture
cap = cv2.VideoCapture(0)

while True:
    ret, frame = cap.read()
    if not ret:
        break

    # Convert frame to RGB for MediaPipe processing
    rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    result = face_mesh.process(rgb)
    now = time.time()

    ear_flag = False
    cnn_flag = False

    # If face landmarks detected, process eyes and run CNN prediction
    if result.multi_face_landmarks:
        landmarks = result.multi_face_landmarks[0].landmark
        left_ear = get_eye_ratio(landmarks, left=True)
        right_ear = get_eye_ratio(landmarks, left=False)
        ear_val = (left_ear + right_ear) / 2.0
        ear_flag = ear_val < EAR_THRESH
        ear_win.append(1 if ear_flag else 0)
```

```
        # Crop face region for CNN input based on landmarks
        h, w, _ = frame.shape
        x_coords = [int(landmarks[i].x * w) for i in range(33, 133)]
        y_coords = [int(landmarks[i].y * h) for i in range(33, 133)]
        x1, x2 = max(min(x_coords)-10, 0), min(max(x_coords)+10, w)
        y1, y2 = max(min(y_coords)-10, 0), min(max(y_coords)+10, h)
        face_crop = frame[y1:y2, x1:x2]

        if face_crop.size:
            inp = transform(face_crop).unsqueeze(0).to(device)
            with torch.no_grad():
                out = model(inp)
                probs = torch.softmax(out, 1)
            conf, pred = probs.max(1)
            cnn_flag = (pred.item() == 1) and (conf.item() > CNN_THRESH)
            cnn_win.append(1 if cnn_flag else 0)

    # Combine EAR and CNN detections over sustained frames
    combined_detect = sum(ear_win) >= SUSTAIN_FRAMES or sum(cnn_win) >= SUSTAIN_FRAMES

    if combined_detect:
        if eyes_closed_start is None:
            eyes_closed_start = now
    else:
        if eyes_closed_start is not None:
            # If eyes closed longer than 3 seconds and first alert given, escalate alert
            if now - eyes_closed_start >= 3 and alert_level == 1:
                alert_level = 2  # Second alert condition met
            else:
                alert_level = 0
        eyes_closed_start = None
        first_warn = False
        second_warn = False
        active_warning_text = None
```

```
    # Set alert messages based on duration of eyes closed
    if eyes_closed_start:
        dur = now - eyes_closed_start
        if dur >= 6 and not second_warn:
            active_warning_text = "!!! SERIOUS ALERT: Pull Over Immediately !!!"
            last_warning_time = now
            second_warn = True
            alert_level = 2
        elif dur >= 3 and not first_warn:
            active_warning_text = "Take a short break!"
            last_warning_time = now
            first_warn = True
            alert_level = 1

    # Display alert message on screen for 3 seconds
    if active_warning_text and now - last_warning_time < 3:
        cv2.putText(frame, active_warning_text, (10, 70), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0,165,255), 2)

    status = 'normal'
    if eyes_closed_start and (now - eyes_closed_start) >= 1:
        status = 'DROWSY'
    elif sum(cnn_win) >= SUSTAIN_FRAMES:
        status = 'DROWSY'

    color = (0,0,255) if status=='DROWSY' else (0,255,0)
    cv2.putText(frame, f"Final: {status}", (10,30), cv2.FONT_HERSHEY_SIMPLEX, 0.8, color, 2)

    cv2.imshow("MediaPipe Drowsiness Detection", frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

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







This script implements a real-time driver drowsiness detection system using a hybrid approach combining:

- MediaPipe Face Mesh: For detecting facial landmarks and calculating the Eye Aspect Ratio (EAR), which estimates eye openness to detect signs of eye closure.
- MobileNetV2-based CNN: Pretrained and fine-tuned model that classifies cropped face images into "drowsy" or "non-drowsy" categories.

Key Components:

- Model Initialization:
    - The DrowsyNet class defines a MobileNetV2 backbone with frozen pretrained weights and a dropout layer before the final classification layer.
    - The pretrained model weights (best_drowsy_model.pth) are loaded and set to evaluation mode.
- Facial Landmark Detection:
    - MediaPipe Face Mesh detects facial landmarks continuously from webcam frames.

    - The get_eye_ratio function computes EAR for both eyes using specific landmark points.
    - EAR thresholding (EAR < 0.25) helps detect eye closure.
- Face Cropping and CNN Inference:
    - The face region is dynamically cropped based on detected landmarks.
    - The cropped image is transformed and passed through the CNN for drowsiness prediction.
    - CNN confidence thresholding (confidence > 0.6) is applied.
- Temporal Smoothing:
    - A deque buffer maintains recent frame results for EAR and CNN predictions (0.5 seconds window).
    - Combined detection logic triggers alerts if sustained eye closure or CNN drowsiness is detected.
- Alert Logic:
    - Two-level alert system:
        - First alert after 3 seconds of continuous drowsiness: "Take a short break!"
        - Second alert after 6 seconds: "!!! SERIOUS ALERT: Pull Over Immediately !!!"
    - Alerts appear on the video feed for 3 seconds.
- User Interface:
    - Real-time webcam feed shows current status: "DROWSY" or "normal" with color-coded text.
    - Pressing 'q' exits the program gracefully.

RESOURCES

[1] D. Perumandla, "Drowsiness Dataset", Kaggle, 2023. [Online]. Available: https://www.kaggle.com/datasets/dheerajperumandla/drowsiness-dataset/data

[2] I. Nasri, "Driver Drowsiness Dataset (DDD)", Kaggle, 2022. [Online]. Available: https://www.kaggle.com/datasets/ismailnasri20/driver-drowsiness-dataset-ddd

[3] Öztürk, E., & Ünver, H. M. (2022). Driver fatigue detection using image processing techniques: A comparison of EAR and PERCLOS methods. Journal of the Institute of Natural Sciences, Sakarya University, 26(1), 190–204.

[4] Perumandla, D. (2019). Driver Drowsiness Detection Using Deep Learning. SSRN Electronic Journal. SSRN ID: 3356401.

[5] Salem, D., & Waleed, M. (2024). Drowsiness detection in real-time via convolutional neural networks and transfer learning. Journal of Engineering and Applied Science, 71, 122.

[6] Karoria, S., Mishra, S., & Verma, P. (2024). Driver Drowsiness Detection with MediaPipe and Deep Learning. International Research Journal of Modernization in Engineering Technology and Science, 6(4).

[7] Vo, H.-T., Ngoc, H. T., & Quach, L. D. (2023). An approach to hyperparameter tuning in transfer learning for driver drowsiness detection based on Bayesian optimization and random search. International Journal of Advanced Computer Science and Applications, 14(4).

[8] Wang, X., Jin, Y., Schmitt, S., & Olhofer, M. (2022). Recent advances in Bayesian optimization. arXiv preprint arXiv:2206.03301.

[9] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L.-C. (2018). MobileNetV2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 4510–4520).

[10] Wei, Y., Wang, M., Chen, Y., & Gao, Z. (2016). A yawn detection method based on multi-feature fusion. In 2016 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER) (pp. 288–291).

[11] Lugaresi, C., Tang, J., Nash, H., McGuire, M., Collado, A., Leigh, J., ... & Cree, M. J. (2019). MediaPipe: A framework for building perception pipelines.