PROJECT REPORT

DEEP LEARNING

"DRIVER DROWSINESS DETECTION"

SUBMITTED BY:-

MEMBER 1:	MEMBER 2:
KAMNA RAIKWAL	BHUMIKA MISHRA
12212043	12212022
CS A3	CSA2
VI SEMESTER	VI SEMESTER

DEPARTMENT OF COMPUTER ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY, KURUKSHETRA

ABSTRACT

With this Python project, we will be creating a real-time web application that detects the state of the driver. We will also compare different types of classifiers and their accuracy on the given dataset.

- ★ Dataset link- https://drive.google.com/drive/folders/1R-YS3-F__1IjIDX2gT3yhZZ1u91nYYXQ?usp=drive_link

For this, we are using the Driver Drowsiness Dataset from Kaggle, which contains RGB images classified into two categories: Drowsy and Non-Drowsy. The dataset has more than 41,790 images, each with a resolution of 227 × 227 pixels. First, the data is loaded, the images are resized to the expected dimensions, then normalized, and the labels are one-hot encoded. The dataset is split into training and test sets with a 80:20 ratio, and then further divided into batches of 32.

➢ The dataset is evaluated using multiple classifiers, and their accuracies and loss are listed below:

1. Custom CNN Model (10 epochs) :-
Train Acc: 0.9988, Train Loss: 0.0069, Val Acc: 1.0000, Val Loss: 0.0012

2. VGG16 (Transfer learning and Fine tuning combined) :-
Train Acc: 1.0000, Train Loss: 1.47e-04, Val Acc: 1.0000, Val Loss: 77e-05

3. Custom FNN using HOG extractor (10 epochs) :-
Train Acc: 0.9965, Train Loss: 0.0237, Val Acc: 0.9982, Val Loss: 0.0089

4. Custom FNN using SIFT extractor (10 epochs) :-
Train Acc: 0.7981, Train Loss: 0.5207, Val Acc: 0.7982, Val Loss: 0.5171

5. FNN using combined HOG + SIFT (10 epochs) :-
Train Acc: 0.9996, Train Loss: 0.0073, Val Acc: 0.9964, Val Loss: 0.0374

6. GoogleNet using Transfer Learning and Fine Tuning combined (10 epochs) :-
Train Acc: 0.9565, Train Loss: 0.1470, Val Acc: 0.9868, Val Loss: 0.0763

7. ResNet using Transfer Learning and Fine Tuning combined (8 epochs) :-
Train Acc: 0.9378, Train Loss: 0.2367, Val Acc: 0.9031, Val Loss: 0.3196

8. DenseNet using Transfer Learning (10 epochs) :-
Train Acc: 0.8943, Train Loss: 0.2870, Val Acc: 0.9330, Val Loss: 0.2340

Further, we used the optimal weights generated from VGG-16 and applied them to a real-time application capturing the frames of the users. The user's video is captured using OpenCV to generate continuous frames. Each frame is preprocessed to extract facial features using a Haar Cascade classifier. Based on the extracted features, the model classifies the user as either Drowsy or Alert with a certain probability. Depending on this probability and a predefined threshold, the user is classified accordingly, and an alarm is triggered if the user is detected as Drowsy.

INTRODUCTION

Background :

Drowsy driving is a major concern in today’s fast-paced world. Many people drive long distances or work irregular shifts, which can lead to fatigue and eventually cause them to doze off while driving. Unfortunately, this can result in serious accidents and fatalities. Over the years, there has been growing interest in finding ways to detect and prevent drowsiness in drivers before it becomes dangerous.

Problem Statement :

The main goal of this project is to design a system that can monitor a driver's face activity in real-time and detect signs of drowsiness. If the driver's eyes remain closed for too long, the system should trigger an alert to wake them up. The solution must be fast, reliable, and able to work in real-time using a simple camera.

Objectives :

- To build and compare different deep learning models for detecting whether driver is feeling drowsy or not.
- To integrate the VGG16 model into a live webcam feed using OpenCV.
- To raise an alert when drowsiness is detected.
- To test both image-based models (like CNNs and VGG16) and feature-based models (using HOG and SIFT with a simple FNN).

Motivation :

Driving under fatigue is one of those hidden dangers we often overlook. If technology can help prevent even a fraction of such accidents, it's worth building. With deep learning becoming more accessible, this project offers a practical and potentially life-saving application.

Overview of Report :

The report begins with a look at previous work done in this area, followed by a detailed explanation of the methodology, the models we used, and how we trained them. Then we share the results and insights from our experiments before wrapping up with conclusions and future directions.

Related Work

A lot of research has been done around drowsiness detection over the years. Some approaches use physiological signals like EEG (brain waves) or ECG (heart rate), which are very accurate but not practical — drivers don’t want to wear sensors on their heads.

Computer vision offers a less invasive and more comfortable alternative.

With the rise of deep learning, models like CNNs have taken the lead. Pretrained networks like **VGG16** offer strong performance, especially when fine-tuned with custom data. On the other hand, traditional feature extraction methods like **HOG** and **SIFT**, when combined with fully connected neural networks (FNNs), can also give good results with less computational cost.

This project combines all of these techniques and compares their performance to find the most suitable one for real-time drowsiness detection.

Methodology

Dataset :

We used a comprehensive dataset designed specifically for drowsiness detection, consisting of **RGB images** categorized into two classes: **Drowsy** and **Non-Drowsy**. The dataset has the following properties:

- **Image Dimensions**: 227 × 227 pixels
- **Total Images**: Over **41,790** labeled samples
- **Image Type**: Colored RGB images

The dataset captures a variety of facial expressions and conditions such as eye closure, providing rich features for training. Its large size helped in training deep learning models effectively while still presenting challenges like class balance and feature variability.

★ Dataset link- https://drive.google.com/drive/folders/1R-YS3-F__1IjIDX2gT3yhZZ1u91nYYXQ?usp=drive_link

Preprocessing:

➢ **Resize**: Images were resized (for custom model and vgg16 and fnn -> 227
for googleNet -> 299
for resNet and denseNet -> 224) to fit model input requirements.
➢ **Normalization**: All pixel values were scaled to the [0, 1] range for stable training.
➢ **One-hot Encoding**: Labels were converted into one-hot vectors for use in classification.
➢ **Different Input Shapes**: Since different models require different input dimensions, we adapted image preprocessing accordingly.

Hardware Utilization :

We configured TensorFlow stream execution to better utilize GPU memory and avoid overloading RAM, especially during training large models or loading high-resolution images. This allowed faster training and smoother model updates.

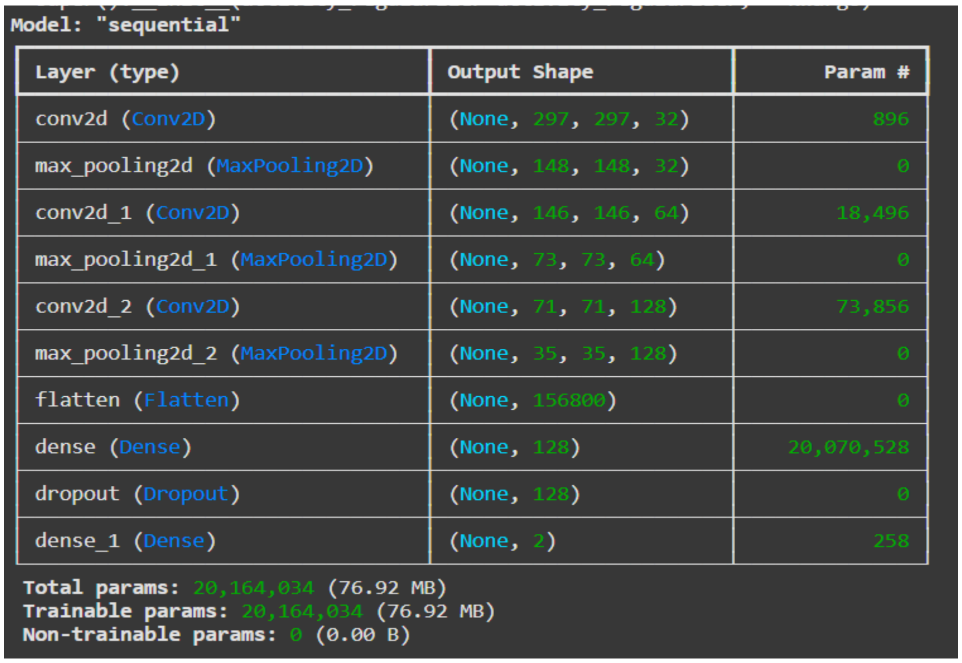
Model Development

We implemented and evaluated on different architectures:

1. Custom CNN (10 Epochs)

A basic convolutional neural network was designed with:

- **Three convolutional layers**, with Relu activation function, each followed by max pooling.
- **Dropout (0.5) and dense layers** for regularization and classification.
- **~20,164k trainable parameters**.
- The models were then trained using categorical-crossentropy loss and optimized using Adam optimizer.



```
Model: "sequential"
┌─────────────────────────────────┬────────────────────────┬───────────────┐
│ Layer (type)                    │ Output Shape           │       Param # │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d (Conv2D)                 │ (None, 297, 297, 32)   │           896 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d (MaxPooling2D)    │ (None, 148, 148, 32)   │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_1 (Conv2D)               │ (None, 146, 146, 64)   │        18,496 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_1 (MaxPooling2D)  │ (None, 73, 73, 64)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_2 (Conv2D)               │ (None, 71, 71, 128)    │        73,856 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_2 (MaxPooling2D)  │ (None, 35, 35, 128)    │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten (Flatten)               │ (None, 156800)         │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                   │ (None, 128)            │    20,070,528 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)               │ (None, 128)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                 │ (None, 2)              │           258 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 20,164,034 (76.92 MB)
 Trainable params: 20,164,034 (76.92 MB)
 Non-trainable params: 0 (0.00 B)
```

2. FNN with Handcrafted Features (10 Epochs)

We extracted features using:

- **HOG (Histogram of Oriented Gradients)**.
- **SIFT (Scale-Invariant Feature Transform)**.
- Also tested a combination of both.

The features were fed into a basic feedforward neural network with:

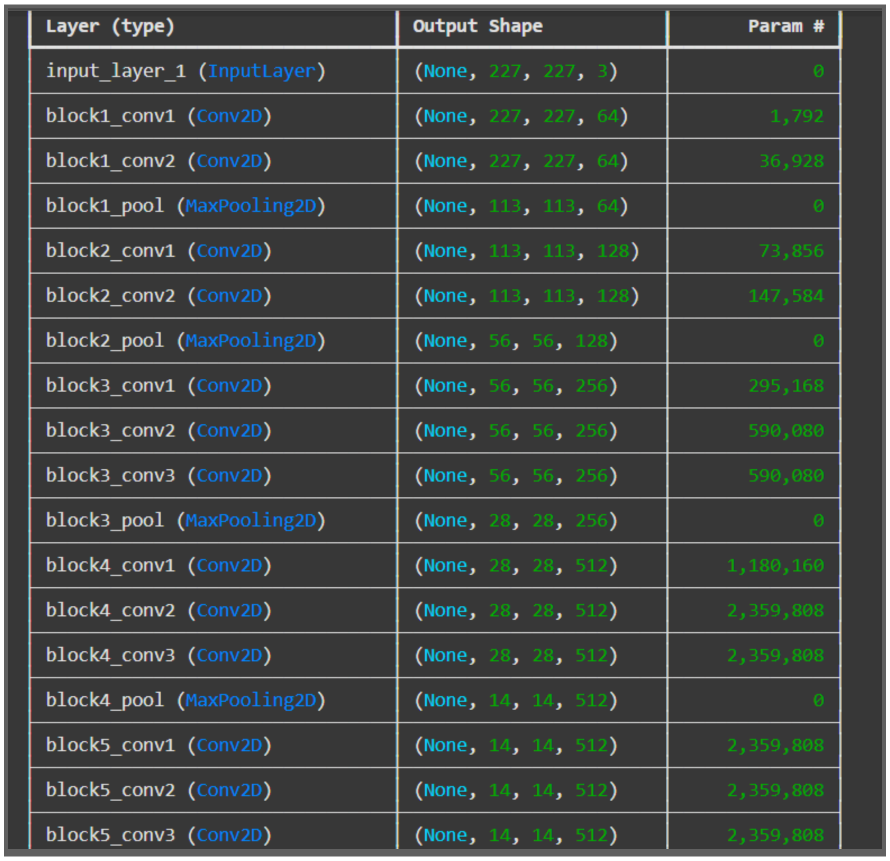
- Two hidden layers.
- ReLU activation and dropout.

Results:

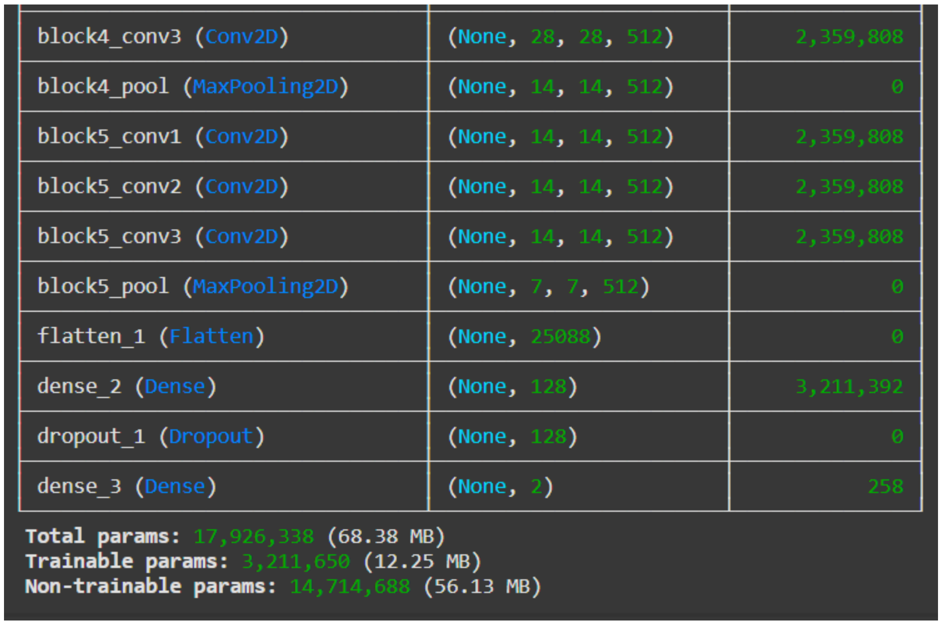
- Less accurate than CNNs or pretrained models.
- However, the FNN was **lightweight and fast**, which could be useful in constrained environments.

3. VGG16 (10 Epochs)

- Applied Transfer Learning for 5 epochs.
- **Last 4 layers were unfreezed** to retain general features.
- Then 5 epochs are in continuation of transfer learning, in fine tuning.
- Trained with our preprocessed dataset using a small learning rate for fine-tuning.



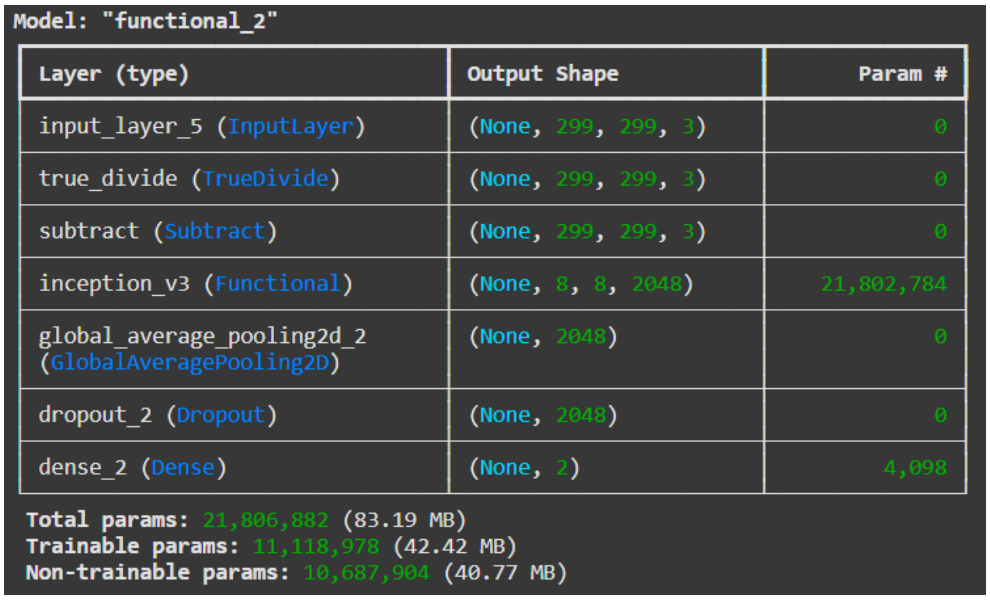
```
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
│ input_layer_1 (InputLayer)      │ (None, 227, 227, 3)    │             0 │
│ block1_conv1 (Conv2D)           │ (None, 227, 227, 64)   │         1,792 │
│ block1_conv2 (Conv2D)           │ (None, 227, 227, 64)   │        36,928 │
│ block1_pool (MaxPooling2D)      │ (None, 113, 113, 64)   │             0 │
│ block2_conv1 (Conv2D)           │ (None, 113, 113, 128)  │        73,856 │
│ block2_conv2 (Conv2D)           │ (None, 113, 113, 128)  │       147,584 │
│ block2_pool (MaxPooling2D)      │ (None, 56, 56, 128)    │             0 │
│ block3_conv1 (Conv2D)           │ (None, 56, 56, 256)    │       295,168 │
│ block3_conv2 (Conv2D)           │ (None, 56, 56, 256)    │       590,080 │
│ block3_conv3 (Conv2D)           │ (None, 56, 56, 256)    │       590,080 │
│ block3_pool (MaxPooling2D)      │ (None, 28, 28, 256)    │             0 │
│ block4_conv1 (Conv2D)           │ (None, 28, 28, 512)    │     1,180,160 │
│ block4_conv2 (Conv2D)           │ (None, 28, 28, 512)    │     2,359,808 │
│ block4_conv3 (Conv2D)           │ (None, 28, 28, 512)    │     2,359,808 │
│ block4_pool (MaxPooling2D)      │ (None, 14, 14, 512)    │             0 │
│ block5_conv1 (Conv2D)           │ (None, 14, 14, 512)    │     2,359,808 │
│ block5_conv2 (Conv2D)           │ (None, 14, 14, 512)    │     2,359,808 │
│ block5_conv3 (Conv2D)           │ (None, 14, 14, 512)    │     2,359,808 │
```



```
│ block4_conv3 (Conv2D)             │ (None, 28, 28, 512)       │     2,359,808 │
│ block4_pool (MaxPooling2D)        │ (None, 14, 14, 512)       │             0 │
│ block5_conv1 (Conv2D)             │ (None, 14, 14, 512)       │     2,359,808 │
│ block5_conv2 (Conv2D)             │ (None, 14, 14, 512)       │     2,359,808 │
│ block5_conv3 (Conv2D)             │ (None, 14, 14, 512)       │     2,359,808 │
│ block5_pool (MaxPooling2D)        │ (None, 7, 7, 512)         │             0 │
│ flatten_1 (Flatten)               │ (None, 25088)             │             0 │
│ dense_2 (Dense)                   │ (None, 128)               │     3,211,392 │
│ dropout_1 (Dropout)               │ (None, 128)               │             0 │
│ dense_3 (Dense)                   │ (None, 2)                 │           258 │
 Total params: 17,926,338 (68.38 MB)
 Trainable params: 3,211,650 (12.25 MB)
 Non-trainable params: 14,714,688 (56.13 MB)
```

4.GoogLeNet :(10 Epochs)

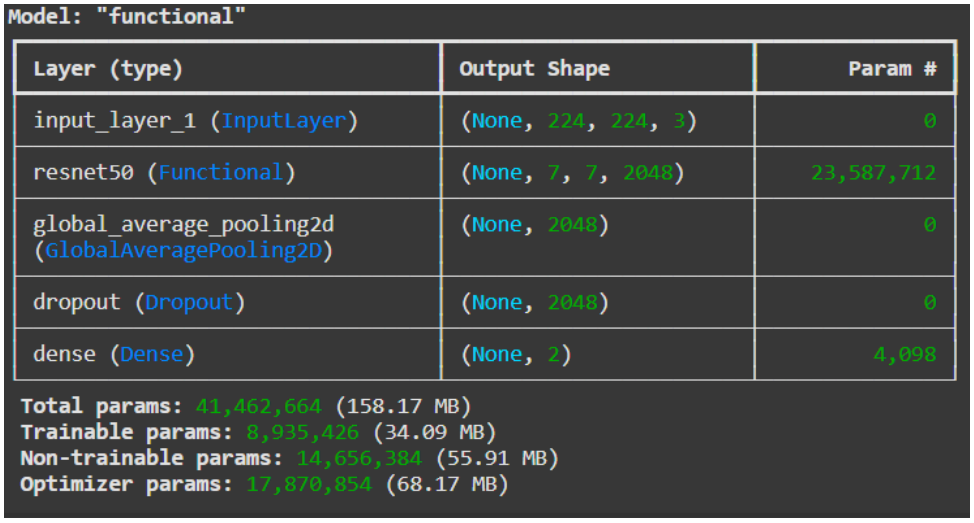
Lightweight and faster than InceptionV3; suitable for real-time applications.



```
Model: "functional_2"
│ Layer (type)                       │ Output Shape              │       Param # │
│ input_layer_5 (InputLayer)         │ (None, 299, 299, 3)       │             0 │
│ true_divide (TrueDivide)           │ (None, 299, 299, 3)       │             0 │
│ subtract (Subtract)                │ (None, 299, 299, 3)       │             0 │
│ inception_v3 (Functional)          │ (None, 8, 8, 2048)        │    21,802,784 │
│ global_average_pooling2d_2         │ (None, 2048)              │             0 │
│ (GlobalAveragePooling2D)           │                           │               │
│ dropout_2 (Dropout)                │ (None, 2048)              │             0 │
│ dense_2 (Dense)                    │ (None, 2)                 │         4,098 │
 Total params: 21,806,882 (83.19 MB)
 Trainable params: 11,118,978 (42.42 MB)
 Non-trainable params: 10,687,904 (40.77 MB)
```

5.ResNet: (8 Epochs)

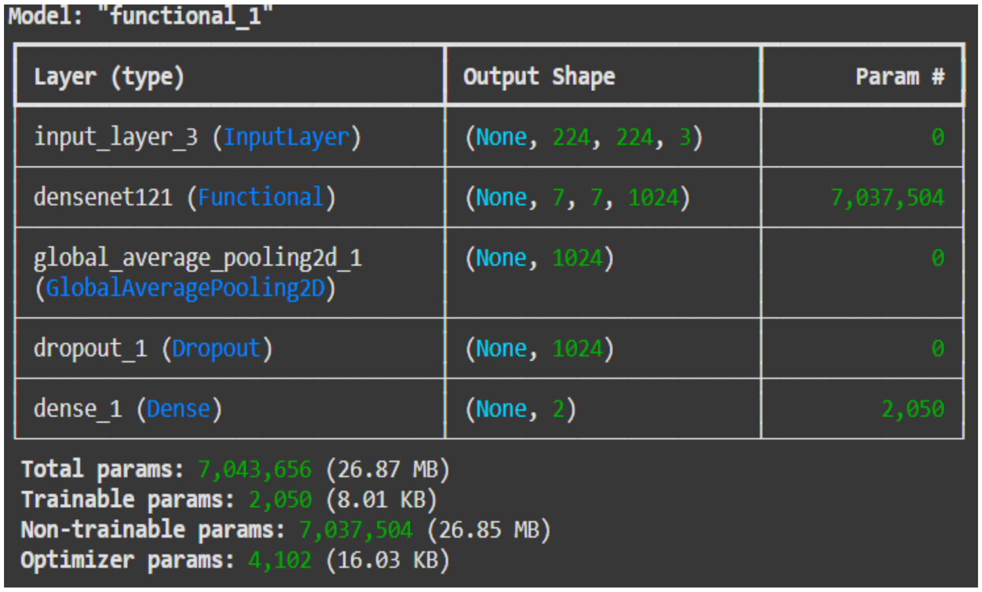
Introduced residual connections to combat vanishing gradient issues in deep networks.



```
Model: "functional"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                       ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input_layer_1 (InputLayer)         │ (None, 224, 224, 3)    │             0 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ resnet50 (Functional)              │ (None, 7, 7, 2048)     │    23,587,712 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ global_average_pooling2d           │ (None, 2048)           │             0 │
│ (GlobalAveragePooling2D)           │                        │               │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)                  │ (None, 2048)           │             0 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                      │ (None, 2)              │         4,098 │
└────────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 41,462,664 (158.17 MB)
 Trainable params: 8,935,426 (34.09 MB)
 Non-trainable params: 14,656,384 (55.91 MB)
 Optimizer params: 17,870,854 (68.17 MB)
```

6.DenseNet:*(10 Epochs)*

Dense connectivity enabled better feature reuse; improved performance with fewer parameters.



```
Model: "functional_1"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                       ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input_layer_3 (InputLayer)         │ (None, 224, 224, 3)    │             0 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ densenet121 (Functional)           │ (None, 7, 7, 1024)     │     7,037,504 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ global_average_pooling2d_1         │ (None, 1024)           │             0 │
│ (GlobalAveragePooling2D)           │                        │               │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_1 (Dropout)                │ (None, 1024)           │             0 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                    │ (None, 2)              │         2,050 │
└────────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 7,043,656 (26.87 MB)
 Trainable params: 2,050 (8.01 KB)
 Non-trainable params: 7,037,504 (26.85 MB)
 Optimizer params: 4,102 (16.03 KB)
```

Experimental Results

Metrics Used :

- **Accuracy**: To evaluate the model's overall performance.
- **Loss**: To track how well the model is minimizing error during training.
- **Precision, Recall, F1-Score**: To evaluate the model's performance, especially in imbalanced class scenarios.

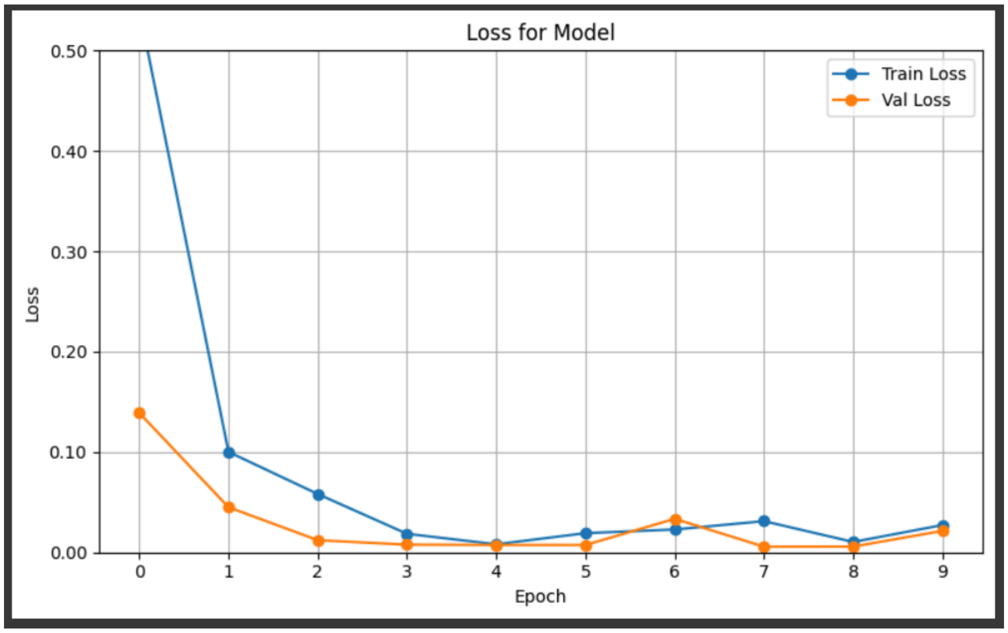
Observations :

- Custom CNN overfitted quickly; adding dropout layer(0.5) helps.
- FNN worked reasonably well with handcrafted features but lacked deep semantic understanding.
- HOG and SIFT require data to be in numpy array, thus increasing RAM usage and inefficient use of GPU.
- Using Fine Tuning in continuation of transfer learning works better.
- The use of **GPU** for preprocessing and **TensorFlow's stream execution** made the training process faster and more efficient.

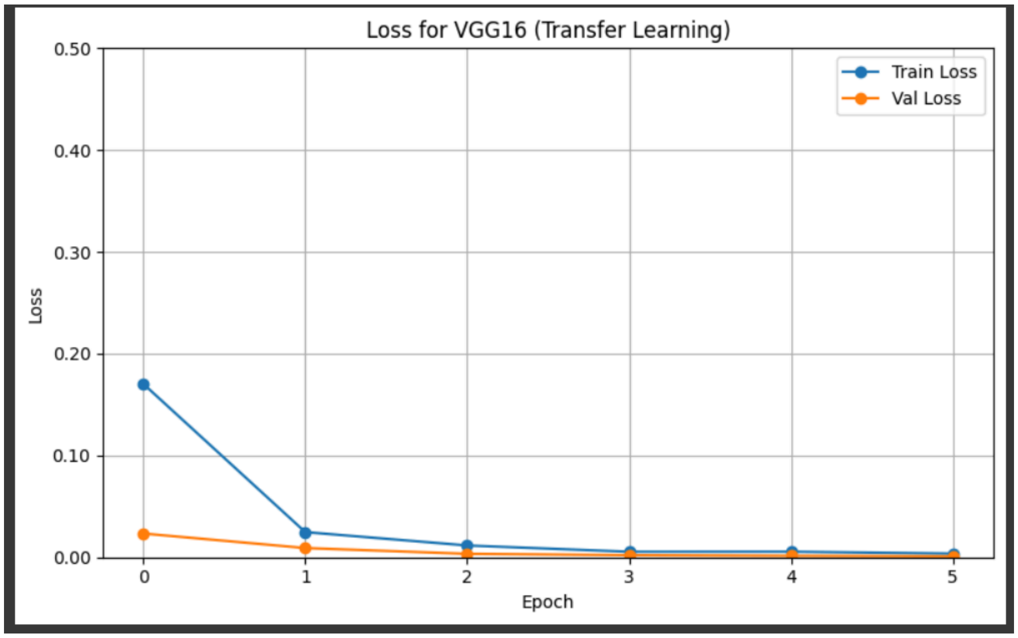
Training Graphs :

➢ **Loss vs. Epoch graphs -**

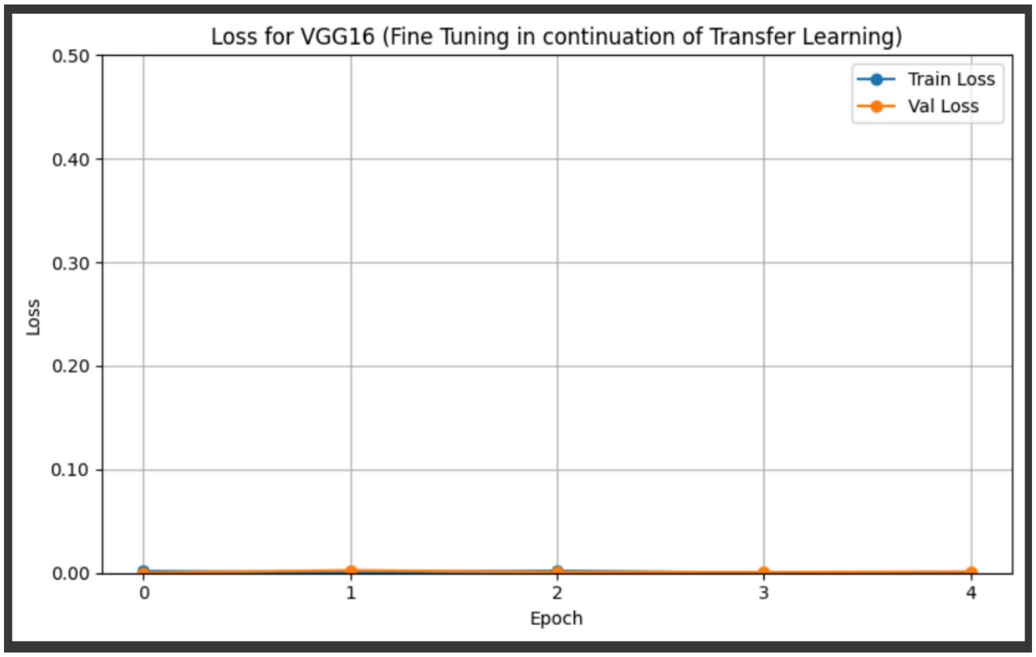
★ CUSTOM MODEL



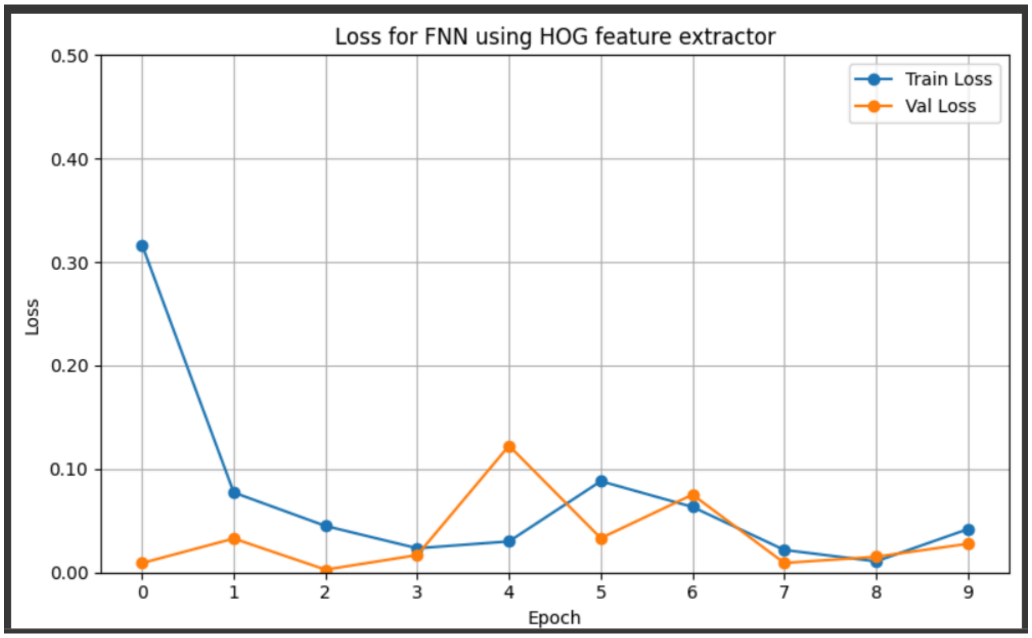
★ VGG 16 Transfer Learning

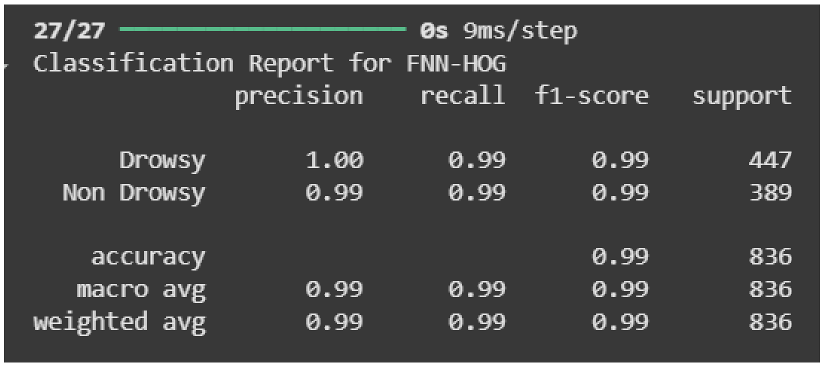


★ FINE TUNING IN CONTINUATION OF TRANSFER LEARNING in VGG16



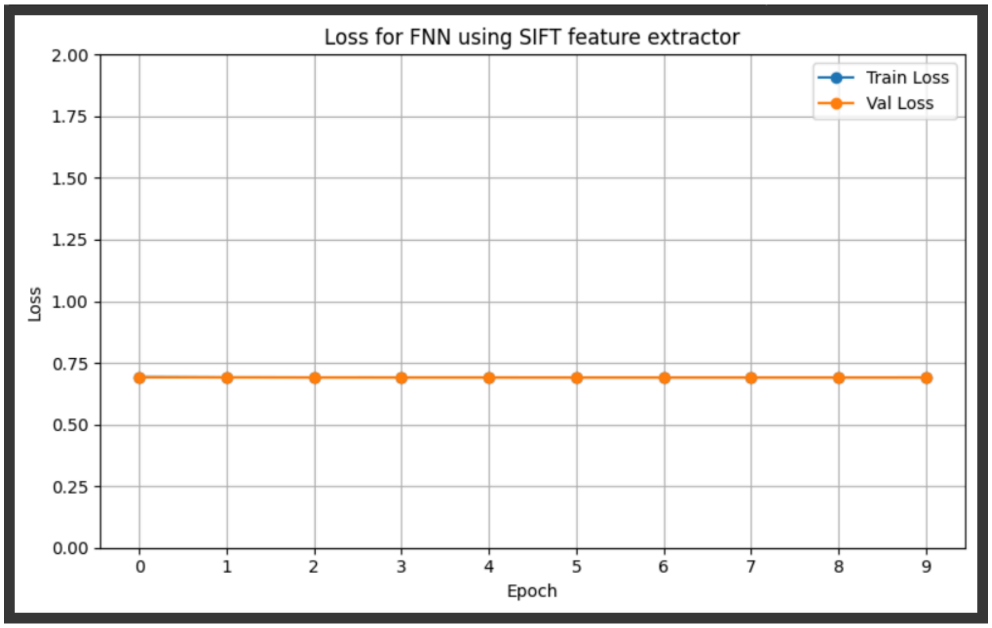
★ FNN USING HOG

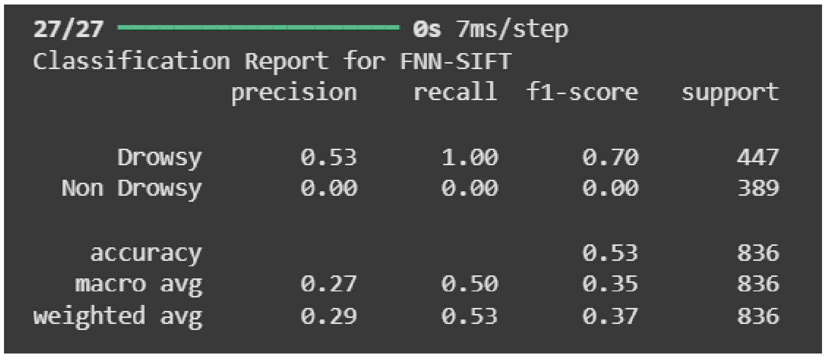




```
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
Classification Report for FNN-HOG
              precision    recall  f1-score   support

      Drowsy       1.00      0.99      0.99       447
  Non Drowsy       0.99      0.99      0.99       389

    accuracy                           0.99       836
   macro avg       0.99      0.99      0.99       836
weighted avg       0.99      0.99      0.99       836
```

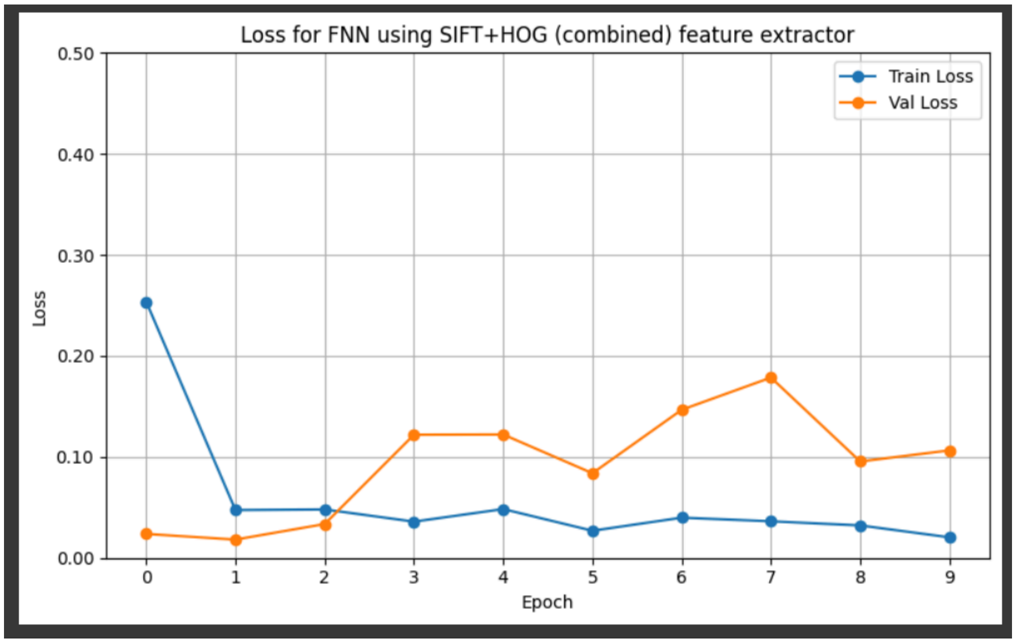
★ FNN USING SIFT

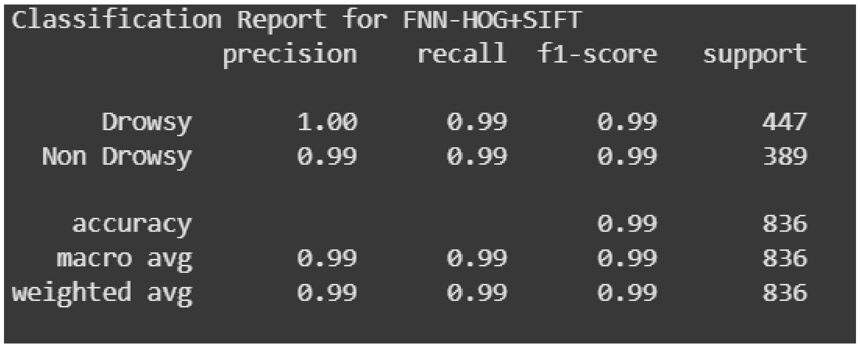




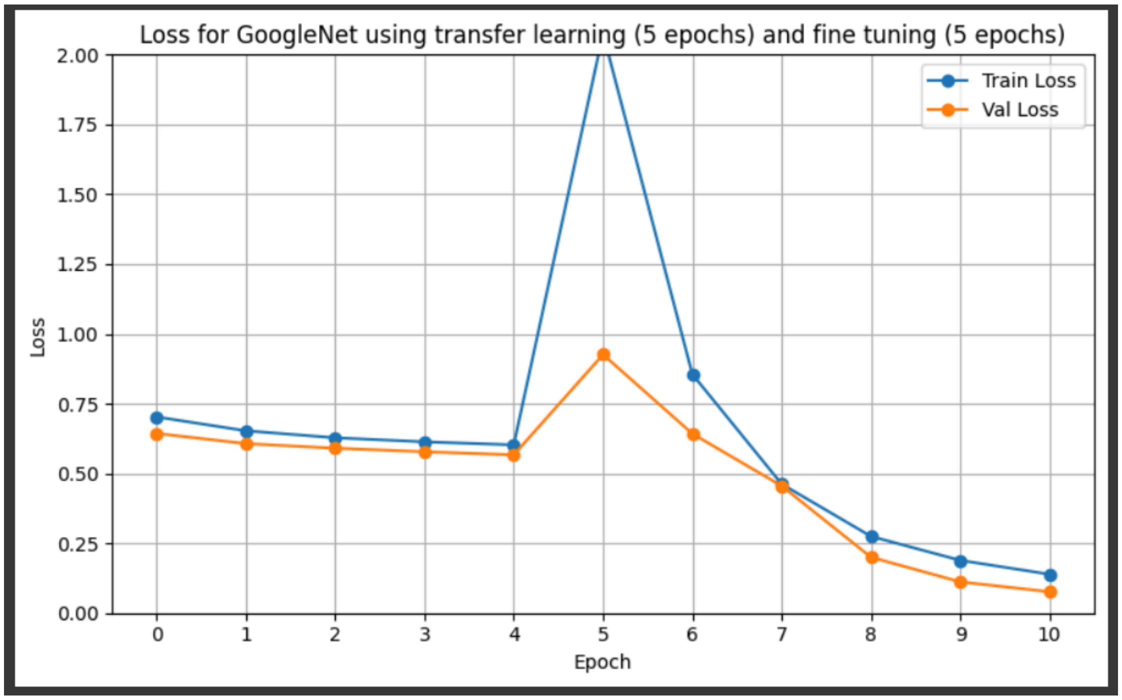
```
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step
Classification Report for FNN-SIFT
              precision    recall  f1-score   support

      Drowsy       0.53      1.00      0.70       447
  Non Drowsy       0.00      0.00      0.00       389

    accuracy                           0.53       836
   macro avg       0.27      0.50      0.35       836
weighted avg       0.29      0.53      0.37       836
```

★ FNN USING (HOG + SIFT)



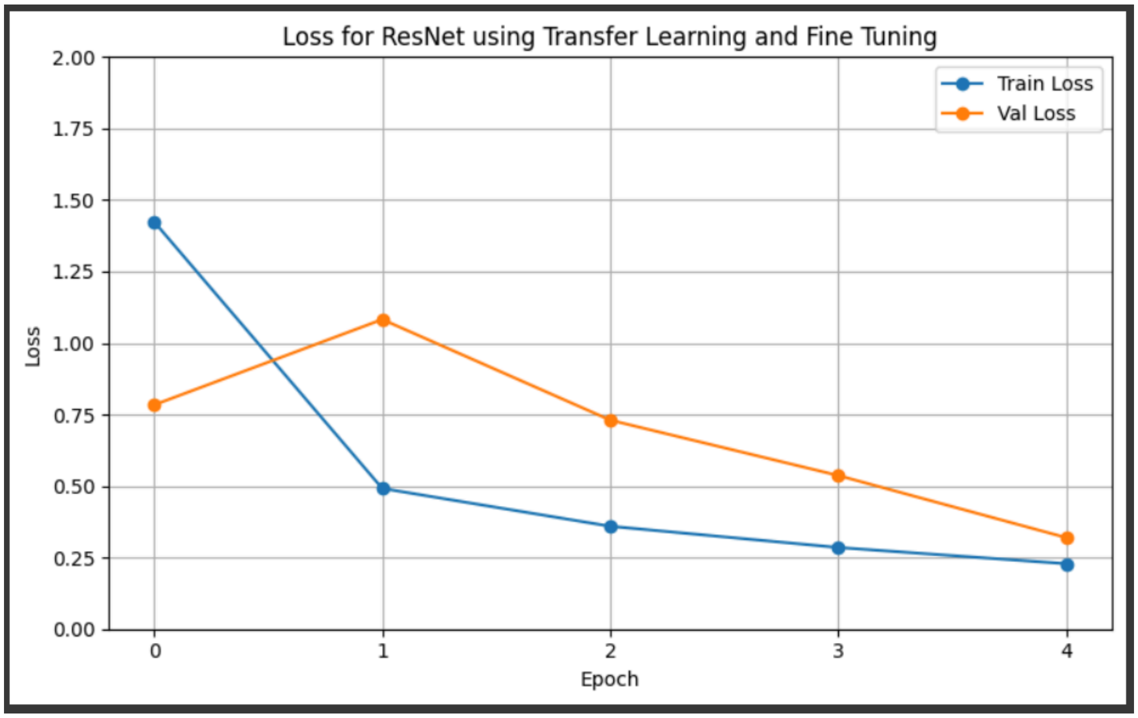

```
Classification Report for FNN-HOG+SIFT
              precision    recall  f1-score   support

      Drowsy       1.00      0.99      0.99       447
  Non Drowsy       0.99      0.99      0.99       389

    accuracy                           0.99       836
   macro avg       0.99      0.99      0.99       836
weighted avg       0.99      0.99      0.99       836
```

★ GOOGLENET USING TRANSFER LEARNING AND FINE TUNING

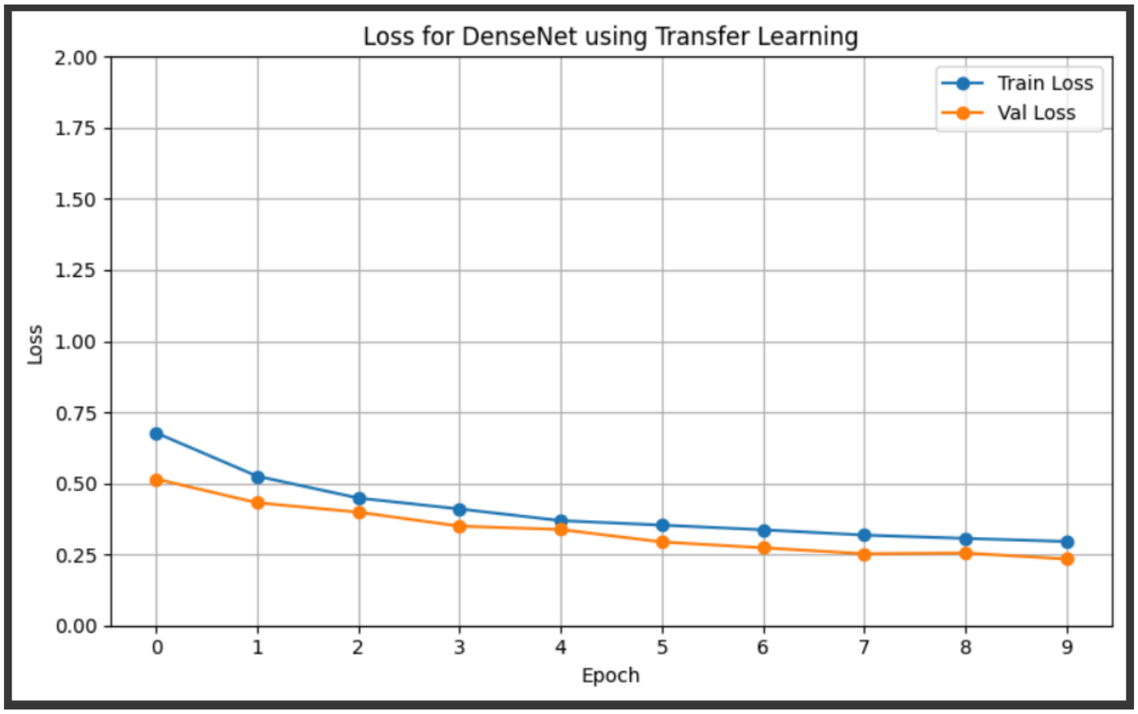


- The sudden spike in the graph is due to the change from transfer learning to fine tuning as the model is trying to adapt it.

★ RESNET USING TRANSFER LEARNING AND FINE TUNING



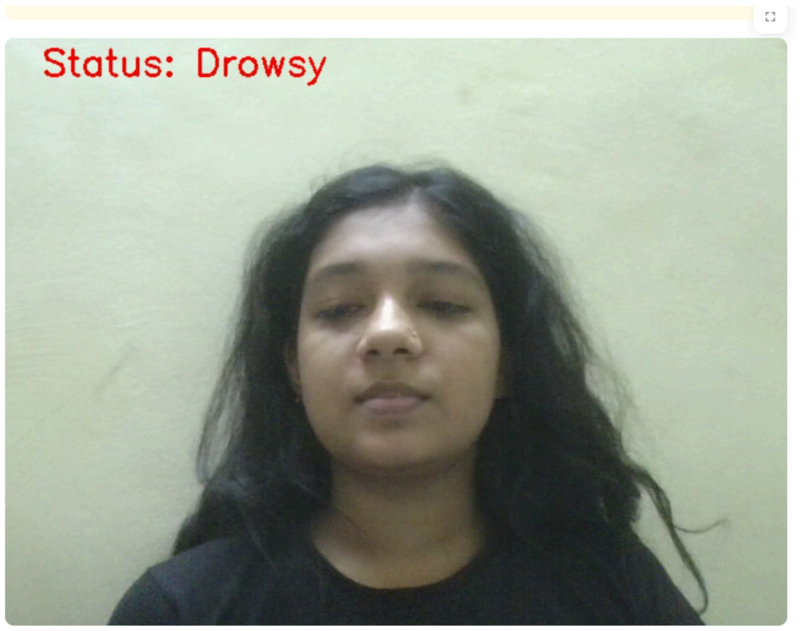
★ DENSENET USING TRANSFER LEARNING



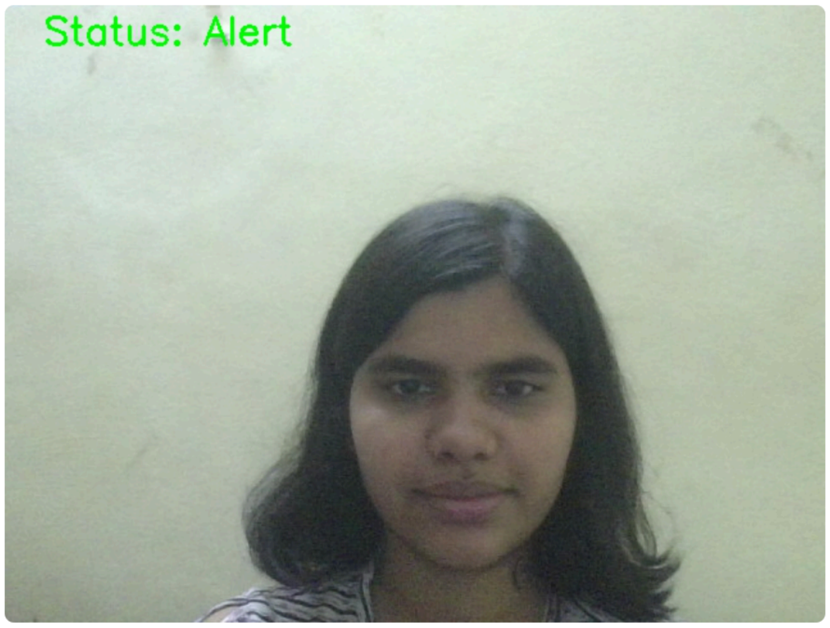
REAL TIME WEB APPLICATION (TESTING)

★ Among all models tested—CNN, FNN, ResNet, DenseNet, and GoogLeNet—VGG16 delivered the best accuracy and consistency. We saved the optimal weights from this model and used them in a real-time web application.
★ We have used a webcam using OpenCV to capture the real time frames of the driver.
★ For preprocessing, we have used Haar Cascade classifier to detect the face and then the same preprocessing is applied as to the training dataset.
★ Frames captured are passed through the models and evaluated as drowsy or alert with some probability.
★ If the driver is drowsy, an alarm is triggered.

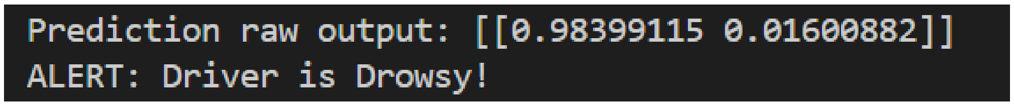
Git Link:-
https://github.com/kamnaraikwal/Driver-Drowsiness-Detection-System



ALERT: Driver is Drowsy!



Driver is Alert!



```
Prediction raw output: [[0.98399115 0.01600882]]
ALERT: Driver is Drowsy!
```

Conclusion

In this project, we successfully implemented and tested multiple approaches for detecting driver drowsiness using image data. While our custom CNN model performed well, it showed clear signs of overfitting. On the other hand, transfer learning with VGG16, combined with fine-tuning the last layers, delivered the best performance with accuracy over 93%.
The use of TensorFlow stream execution allowed efficient training by optimizing GPU usage and reducing memory load. Our FNN model with handcrafted features showed promise for low-resource environments, though it lagged behind in accuracy.

Future Scope:

- Expand dataset with real driving scenarios across lighting conditions.
- Deploy models using IOT devices for real-world applications.